
目錄

前言	1.1
一、 负载均衡和 HTTP 缓存	1.2
1.1 高性能的负载均衡	1.2.1
1.1.0 介绍	1.2.1.1
1.1.1 HTTP 负载均衡	1.2.1.2
1.1.2 TCP 负载均衡配置	1.2.1.3
1.1.3 负载均衡算法	1.2.1.4
1.1.4 限制连接	1.2.1.5
1.2 智能的会话持久化	1.2.2
1.2.0 介绍	1.2.2.1
1.2.1 绑定 Cookie 到服务器	1.2.2.2
1.3 服务器健康监控	1.2.3
1.3.0 介绍	1.2.3.1
1.3.1 健康监控内容	1.2.3.2
1.3.3 TCP 服务器监控检测	1.2.3.3
1.3.4 HTTP 服务器监控检测	1.2.3.4
1.5 大规模可伸缩缓存配置	1.2.4
1.5.0 介绍	1.2.4.1
1.5.1 缓存区域配置(Caching Zones)	1.2.4.2
1.5.2 配置缓存哈希键名	1.2.4.3
1.5.3 跳过被缓存内容	1.2.4.4
1.5.4 缓存性能	1.2.4.5
1.9 UDP 负载均衡	1.2.5
1.9.0 介绍	1.2.5.1
1.9.1 Stream 指令上下文	1.2.5.2
1.9.2 负载均衡算法	1.2.5.3
1.9.3 UDP 服务器健康检测	1.2.5.4
二、 服务器安全与可访问性	1.3
2.11 可访问性控制	1.3.1
2.11.0 介绍	1.3.1.1

2.11.1 基于 IP 地址访问配置	1.3.1.2
2.11.2 跨域资源共享控制	1.3.1.3
2.12 访问限制	1.3.2
2.12.0 介绍	1.3.2.1
2.12.1 限制连接数	1.3.2.2
2.12.2 限制上传下载速度	1.3.2.3
2.12.3 限制带宽	1.3.2.4
2.13 数据加密	1.3.3
2.13.0 介绍	1.3.3.1
2.13.1 客户端加密	1.3.3.2
2.13.2 Upstream 模块加密	1.3.3.3
2.20 实战加密技巧	1.3.4
2.20.1 HTTPS 重定向	1.3.4.1
2.20.3 启用 HTTP 严格传输加密功能	1.3.4.2
三、部署和运维	1.4
3.29 访问日志、错误日志和请求调用栈的调试和问题跟踪	1.4.1
3.29.1 介绍	1.4.1.1
3.29.1 配置访问日志	1.4.1.2
3.29.2 配置错误日志	1.4.1.3
3.29.3 将日志记录到 syslog	1.4.1.4
3.29.4 请求调用栈	1.4.1.5
3.30 性能调优	1.4.2
3.30.0 介绍	1.4.2.1
3.30.1 使用负载测试工具实现自动化测试	1.4.2.2
3.30.2 启用客户端长连接	1.4.2.3
3.30.3 启用 upstream 模块长连接	1.4.2.4
3.30.4 启用响应缓冲区	1.4.2.5
3.30.5 启用访问日志缓冲区	1.4.2.6
3.30.6 操作系统调优	1.4.2.7

Nginx 烹调书

译者：柳公子 huliuqing1989@gmail.com

本书是「[Complete Nginx Cookbook](#)」一书的部分中英文对照翻译版本。

原书以抛出问题，提出解决方案和归纳总结的行文方式，讲解如何配置缓存，负载均衡，安全配置，**WAF**，云服务器部署和其它 **NGINX** 的重要特性。

翻译工具：[有道翻译](#)

Part I: Load Balancing and HTTP Caching

This is Part I of III of NGINX Cookbook. This book is about NGINX the web server, reverse proxy, load balancer, and HTTP cache. Part I will focus mostly on the load-balancing aspect and the advanced features around load balancing, as well as some information around HTTP caching. This book will touch on NGINX Plus, the licensed version of NGINX that provides many advanced features, such as a real-time monitoring dashboard and JSON feed, the ability to add servers to a pool of application servers with an API call, and active health checks with an expected response. The following chapters have been written for an audience that has some understanding of NGINX, modern web architectures such as n-tier or microservice designs, and common web protocols such as TCP, UDP, and HTTP. I wrote this book because I believe in NGINX as the strongest web server, proxy, and load balancer we have. I also believe in NGINX's vision as a company. When I heard Owen Garrett, head of products at NGINX, Inc. explain that the core of the NGINX system would continue to be developed and open source, I knew NGINX, Inc. was good for all of us, leading the World Wide Web with one of the most powerful software technologies to serve a vast number of use cases. Throughout this book, there will be references to both the free and open source NGINX software, as well as the commercial product from NGINX, Inc., NGINX Plus. Features and directives that are only available as part of the paid subscription to NGINX Plus will be denoted as such. Most readers in this audience will be users and

advocates for the free and open source solution; this book's focus is on just that, free and open source NGINX at its core. However, this first part provides an opportunity to view some of the advanced features available in the paid solution, NGINX Plus.

1.0 Introduction

Today's internet user experience demands performance and uptime.

To achieve this, multiple copies of the same system are run, and the load is distributed over them. As load increases, another copy of the system can be brought online. The architecture technique is called horizontal scaling. Software-based infrastructure is increasing in popularity because of its flexibility, opening up a vast world of possibility. Whether the use case is as small as a set of two for high availability or as large as thousands world wide, there's a need for a load-balancing solution that is as dynamic as the infrastructure. NGINX fills this need in a number of ways, such as HTTP, TCP, and UDP load balancing, the last of which is discussed in Chapter 9.

This chapter discusses load-balancing configurations for HTTP and TCP in NGINX. In this chapter, you will learn about the NGINX load-balancing algorithms, such as round robin, least connection, least time, IP hash, and generic hash. They will aid you in distributing load in ways more useful to your application. When balancing load, you also want to control the amount of load being served to the application server, which is covered in Recipe 1.4.

1.1.0 介绍

如今的互联网产品的用户体验依赖与产品的可用时间和服务性能。为实现这些目标，多个相同服务被部署在物理设备上构建称服务集群，并使用负载均衡实现请求分发；随着负载的增加，新的服务会被部署到到集群中；这种技

术称之为水平扩展。由于其灵活性，基于软件的可扩展技术越来越受到青睐。而无论是仅有两台服务器的高可用技术方案，还是成千上万台服务器的高可用集群，它们的可用性都需要灵活的负载均衡解决方案才能得以保障。NGINX 提供了多种协议的负载均衡解决方案如：HTTP、TCP 和 UDP 负载均衡，其中 UDP 负载均衡将在第 9 章讲解。

本章将讨论 HTTP 和 TCP 负载均衡技术。还将学习负载均衡算法如：轮询，最少连接数，最短响应时间，IP 哈希和普通哈希。这些负载均衡算法有助于您在项目中选择行之有效的请求分配策略。此外，还将讲解如何将更多的请求分配到那些服务器性能更好的机器上。

1.0 Introduction

Today's internet user experience demands performance and uptime.

To achieve this, multiple copies of the same system are run, and the load is distributed over them. As load increases, another copy of the system can be brought online. The architecture technique is called horizontal scaling. Software-based infrastructure is increasing in popularity because of its flexibility, opening up a vast world of possibility. Whether the use case is as small as a set of two for high availability or as large as thousands world wide, there's a need for a load-balancing solution that is as dynamic as the infrastructure. NGINX fills this need in a number of ways, such as HTTP, TCP, and UDP load balancing, the last of which is discussed in Chapter 9.

This chapter discusses load-balancing configurations for HTTP and TCP in NGINX. In this chapter, you will learn about the NGINX load-balancing algorithms, such as round robin, least connection, least time, IP hash, and generic hash. They will aid you in distributing load in ways more useful to your application. When balancing load, you also want to control the amount of load being served to the application server, which is covered in Recipe 1.4.

1.1.0 介绍

如今的互联网产品的用户体验依赖与产品的可用时间和服务性能。为实现这些目标，多个相同服务被部署在物理设备上构建称服务集群，并使用负载均衡实现请求分发；随着负载的增加，新的服务会被部署到到集群中；这种技

术称之为水平扩展。由于其灵活性，基于软件的可扩展技术越来越受到青睐。而无论是仅有两台服务器的高可用技术方案，还是成千上万台服务器的高可用集群，它们的可用性都需要灵活的负载均衡解决方案才能得以保障。NGINX 提供了多种协议的负载均衡解决方案如：HTTP、TCP 和 UDP 负载均衡，其中 UDP 负载均衡将在第 9 章讲解。

本章将讨论 HTTP 和 TCP 负载均衡技术。还将学习负载均衡算法如：轮询，最少连接数，最短响应时间，IP 哈希和普通哈希。这些负载均衡算法有助于您在项目中选择行之有效的请求分配策略。此外，还将讲解如何将更多的请求分配到那些服务器性能更好的机器上。

1.1 HTTP Load Balancing

Problem

You need to distribute load between two or more HTTP servers.

问题

将用户请求分发到 2 台以上 HTTP 服务器。

Solution

Use NGINX's HTTP module to load balance over HTTP servers

using the upstream block:

```
upstream backend {
    server 10.10.12.45:80 weight=1;
    server app.example.com:80 weight=2;
}
server {
    location / {
        proxy_pass http://backend;
    }
}
```

This configuration balances load across two HTTP servers on port

1. The weight parameter instructs NGINX to pass twice as many connections to the second server, and the weight parameter defaults to 1.

解决方案

使用 NGINX 的 HTTP 模块，将请求分发到有 upstream 块级指令代理的 HTTP 服务器集群，实现负载均衡：

```
upstream backend {
    server 10.10.12.45:80 weight=1;
    server app.example.com:80 weight=2;
}
server {
    location / {
        proxy_pass http://backend;
    }
}
```

配置中启用了两台默认 80 端口 HTTP 服务器构成服务器集群。weight 参数表示没三个请求将有 2 个请求分发到 app.example.com:80 服务器，它的默认值为 1。

Discussion

The HTTP upstream module controls the load balancing for HTTP.

This module defines a pool of destinations, either a list of Unix sockets, IP addresses, and DNS records, or a mix. The upstream module also defines how any individual request is assigned to any of the upstream servers.

Each upstream destination is defined in the upstream pool by the server directive. The server directive is provided a Unix socket, IP address, or an FQDN, along with a number of optional parameters. The optional parameters give more control over the routing of requests. These parameters include the weight of the server in the balancing algorithm; whether the server is in standby mode, available, or unavailable; and how to determine if the server is unavailable. NGINX Plus provides a number of other convenient parameters like connection limits to the server, advanced DNS resolution control, and the ability to slowly ramp up connections to a server after it starts.

结论

HTTP 模块的 `upstream` 用于设置被代理的 HTTP 服务器实现负载均衡。模块内定义一个目标服务器连接池，它可以是 UNIX 套接字、IP 地址、DNS 记录或它们的混合使用配置；此外 `upstream` 还可以通过 `weight` 参数配置，如何分发请求到应用服务器。

所有 HTTP 服务器在 `upstream` 块级指令中由 `server` 指令配置完成。`server` 指令接收 UNIX 套接字、IP 地址或 FQDN(Fully Qualified Domain Name: 全限定域名) 及一些可选参数。可选参数能够精细化控制请求分发。它们包括用于负载均衡算法的 `weight` 参数；判断目标服务器是否可用，及如何判断服务器可用性的 `max_fails` 指令和 `fail_timeout` 指令。NGINX Plus 版本提供了许多其他方便的参数，比如服务器的连接限制、高级DNS解析控制，以及在服务器启动后缓慢地连接到服务器的能力。

1.2 TCP Load Balancing

Problem

You need to distribute load between two or more TCP servers.

问题

将请求分发到 2 台以上 TCP 服务器。

Solution

Use NGINX's stream module to load balance over TCP servers

using the upstream block:

```
stream {
    upstream mysql_read {
        server read1.example.com:3306 weight=5;
        server read2.example.com:3306;
        server 10.10.12.34:3306 backup;
    }
    server {
        listen 3306;
        proxy_pass mysql_read;
    }
}
```

The server block in this example instructs NGINX to listen on TCP port 3306 and balance load between two MySQL database read replicas, and lists another as a backup that will be passed traffic if the primaries are down.

解决方案

在 NGINX 的 stream 模块内使用 upstream 块级指令实现多台 TCP 服务器负载均衡：

```
stream {
    upstream mysql_read {
        server read1.example.com:3306 weight=5;
        server read2.example.com:3306;
        server 10.10.12.34:3306 backup;
    }
    server {
        listen 3306;
        proxy_pass mysql_read;
    }
}
```

例中的 **server** 块级指令指定 NGINX 监听 3306 端口的多台 MySQL 数据库实现负载均衡，其中 10.10.12.34:3306 作为备用数据库服务器当负载均衡请求分发失败时会被启用。

Discussion

TCP load balancing is defined by the NGINX stream module. The stream module, like the HTTP module, allows you to define upstream pools of servers and configure a listening server. When configuring a server to listen on a given port, you must define the port it's to listen on, or optionally, an address and a port. From there a destination must be configured, whether it be a direct reverse proxy to another address or an upstream pool of resources.

The upstream for TCP load balancing is much like the upstream for HTTP, in that it defines upstream resources as servers, configured with Unix socket, IP, or FQDN; as well as server weight, max number of connections, DNS resolvers, and connection ramp-up periods; and if the server is active, down, or in backup mode.

NGINX Plus offers even more features for TCP load balancing.

These advanced features offered in NGINX Plus can be found throughout Part I of this book. Features available in NGINX Plus,

such as connection limiting, can be found later in this chapter. Health checks for all load balancing will be covered in Chapter 2. Dynamic reconfiguration for upstream pools, a feature available in NGINX Plus, is covered in Chapter 8.

结论

TCP 负载均衡在 `stream` 模块中配置实现。`stream` 模块类似于 `http` 模块。配置时需要在 `server` 块中使用 `listen` 指令配置待监听端口或 IP 加端口。接着，需要明确配置目标服务，目标服务可以使代理服务或 `upstream` 指令所配置的连接池。TCP 负载均衡实现中的 `upstream` 指令配置和 HTTP 负载均衡实现中的 `upstream` 指令配置相似。TCP 服务器在 `server` 指令中配置，格式同样为 UNIX 套接字、IP 地址或 FQDN(Fully Qualified Domain Name: 全限定域名)；用于精细化控制的 `weight` 权重参数、最大连接数、DNS 解析器、判断服务是否可用和启用为备选服务的 `backup` 参数一样能在 TCP 负载均衡中使用。NGINX Plus offers even more features for TCP load balancing. These advanced features offered in NGINX Plus can be found throughout Part I of this book. Features available in NGINX Plus, such as connection limiting, can be found later in this chapter. Health checks for all load balancing will be covered in Chapter 2. Dynamic reconfiguration for upstream pools, a feature available in NGINX Plus, is covered in Chapter 8.

1.3 负载均衡算法(Load-Balancing Methods)

Problem

Round-robin load balancing doesn't fit your use case because you have heterogeneous workloads or server pools.

问题

对于负载压力不均匀的应用服务器或服务器连接池，轮询(round-robin)负载均衡算法无法满足业务需求。

Solution

Use one of NGINX's load-balancing methods, such as least connections, least time, generic hash, or IP hash:

```
upstream backend {
    least_conn;
    server backend.example.com;
    server backend1.example.com;
}
```

This sets the load-balancing algorithm for the backend upstream pool to be least connections. All load-balancing algorithms, with the exception of generic hash, will be standalone directives like the preceding example. Generic hash takes a single parameter, which can be a concatenation of variables, to build the hash from.

解决方案

使用 NGINX 提供的其它负载均衡算法，如：最少连接数(least connections)、最短响应时间(least time)、通用散列算法(generic hash)或 IP 散列算法(IP hash)：


```
upstream backend {  
    least_conn;  
    server backend.example.com;  
    server backend1.example.com;  
}
```

上面的 `least_conn` 指令为 `upstream` 所负载的后端服务，指定采用最少连接数负载均衡算法实现负载均衡。所有的负载均衡算法指令，除了通用算列指令外，都和上面示例一样是一个普通指令，需要独占一行配置。通用散列指令，接收一个参数，也可以使一系列变量值的拼接结果，来构建散列值。

Discussion

Not all requests or packets carry an equal weight. Given this, round robin, or even the weighted round robin used in examples prior, will not fit the need of all applications or traffic flow. NGINX provides a number of load-balancing algorithms that can be used to fit particular use cases. These load-balancing algorithms or methods can not only be chosen, but also configured. The following load-balancing methods are available for upstream HTTP, TCP, and UDP pools:

结论

在负载均衡中，并非所有的请求和数据包请求都具有相同的权重。有鉴于此，如上例所示的轮询或带有权重的轮询负载均衡算法，可能并不能满足我们的应用或负载需求。NGINX 提供了一系列的负载均衡算法，以满足不同的运用场景。所有提供的负载均衡算法都可以针对业务场景随意选择和配置，并且都可以应用于 `upstream` 块级指令中的 HTTP、TCP 和 UDP 负载均衡服务器连接池。

Round robin

The default load-balancing method, which distributes requests in order of the list of servers in the upstream pool. Weight can be taken into consideration for a weighted round robin, which could be used if the capacity of the upstream servers varies. The higher the integer value for the weight, the more favored the server will be in the round robin. The algorithm behind weight is simply statistical probability of a weighted average. Round robin is the default load-balancing algorithm and is used if no other algorithm is specified.

轮询负载均衡算法(Round robin)

NGINX 服务器默认的负载均衡算法，该算法将请求分发到 upstream 指令块中配置的应用服务器列表中的任意一个服务器。可以通过应用服务器的负载能力，为应用服务器指定不同的分发权重(weight)。权重的值设置的越大，将被分发更多的请求访问。权重算法的核心技术是，依据访问权重求均值进行概率统计。轮询作为默认的负载均衡算法，将在没有指定明确的负载均衡指令的情况下启用。

Least connections

Another load-balancing method provided by NGINX. This method balances load by proxying the current request to the upstream server with the least number of open connections proxied through NGINX. Least connections, like round robin, also takes weights into account when deciding to which server to send the connection. The directive name is least_conn.

最少连接数负载均衡算法(Least connections)

NGINX 服务器提供的另一个负载均衡算法。它会将访问请求分发到

upstream 所代理的应用服务器中，当前打开连接数最少的应用服务器

实现负载均衡。最少连接数负载均衡，提供类似轮询的权重选项，来决定

给性能更好的应用服务器分配更多的访问请求。该指令的指令名称是

least_conn。

Least time

Available only in NGINX Plus, is akin to least connections in

that it proxies to the upstream server with the least number of

current connections but favors the servers with the lowest aver-

age response times. This method is one of the most sophistica-

ted load-balancing algorithms out there and fits the need of

highly performant web applications. This algorithm is a value

add over least connections because a small number of connec-

tions does not necessarily mean the quickest response. The

directive name is least_time.

最短响应时间负载均衡算法(least time)

该算法仅在 NGINX PLUS 版本中提供，和最少连接数算法类似，它将请求

分发给平均响应时间更短的应用服务器。它是负载均衡算法最复杂的算法

之一，能够适用于需要高性能的 Web 服务器负载均衡的业务场景。该算法

是对最少连接数负载均衡算法的优化实现，因为最少的访问连接并非意味着

更快的响应。该指令的配置名称是 least_time。

Generic hash

The administrator defines a hash with the given text, variables

of the request or runtime, or both. NGINX distributes the load

amongst the servers by producing a hash for the current request

and placing it against the upstream servers. This method is very

useful when you need more control over where requests are sent or determining what upstream server most likely will have the data cached. Note that when a server is added or removed from the pool, the hashed requests will be redistributed. This algorithm has an optional parameter, `consistent`, to minimize the effect of redistribution. The directive name is `hash`.

通用散列负载均衡算法(**Generic hash**)

服务器管理员依据请求或运行时提供的文本、变量或文本和变量的组合来生成散列值。通过生成的散列值决定使用哪一台被代理的应用服务器，并将请求分发给它。在需要对访问请求进行负载可控，或将访问请求负载到已经有数据缓存的应用服务器的业务场景下，该算法会非常有用。需要注意的是，在 `upstream` 中有应用服务器被加入或删除时，会重新计算散列进行分发，因而，该指令提供了一个可选的参数选项来保持散列一致性，减少因应用服务器变更带来的负载压力。该指令的配置名称是 `hash`。

IP hash

Only supported for HTTP, is the last of the bunch. IP hash uses the client IP address as the hash. Slightly different from using the remote variable in a generic hash, this algorithm uses the first three octets of an IPv4 address or the entire IPv6 address. This method ensures that clients get proxied to the same upstream server as long as that server is available, which is extremely helpful when the session state is of concern and not handled by shared memory of the application. This method also takes the weight parameter into consideration when distributing the hash. The directive name is `ip_hash`.

IP 散列负载均衡算法(IP hash)

该算法仅支持 HTTP 协议，它通过计算客户端的 IP 地址来生成散列值。

不同于采用请求变量的通用散列算法，IP 散列算法通过计算 IPv4 的前三个八进制位或整个 IPv6 地址来生成散列值。这对需要存储使用会话，而又没有使用共享内存存储会话的应用服务来说，能够保证同一个客户端请求，在应用服务可用的情况下，永远被负载到同一台应用服务器上。

该指令同样提供了权重参数选项。该指令的配置名称是 `ip_hash`。

2.0 Introduction

While HTTP may be a stateless protocol, if the context it's to convey were stateless, the internet would be a much less interesting place. Many modern web architectures employ stateless application tiers, storing state in shared memory or databases. However, this is not the reality for all. Session state is immensely valuable and vast in interactive applications. This state may be stored locally for a number of reasons; for example, in applications where the data being worked is so large that network overhead is too expensive in performance. When state is stored locally to an application server, it is extremely important to the user experience that the subsequent requests continue to be delivered to the same server. Another portion of the problem is that servers should not be released until the session has finished. Working with stateful applications at scale requires an intelligent load balancer. NGINX Plus offers multiple ways to solve this problem by tracking cookies or routing. NGINX Plus's sticky directive alleviates difficulties of server affinity at the traffic controller, allowing the application to focus on its core. NGINX tracks session persistence in three ways: by creating and tracking its own cookie, detecting when applications prescribe cookies, or routing based on runtime variables.

2.0 简介

尽管 HTTP 协议是无状态的协议，但如果互联网访问都是基于无状态的访问的话，这个世界将会索然无味。但现代的 Web 架构大多基于一个无状态的应用架构，而

将会话(session)存储到内存或数据库中。可惜，这并非全部的事实。会话对于应用的交互来说功能显著，作用明显。在设计是可能出于一系列的因素考虑，将会话存储到应用服务器中；其中之一就是需要存储的会话数据太多，导致网络开销太大，将会话存储在本地服务器，能显著提升能将用户请求分发到同一台服务器的用户体验；另个方面的考虑是，如果会话没有被销毁，服务器则不应被释放掉。在具有高负载的应用服务器上使用会话，需要更加智能的负载均衡解决方案。

2.0 Introduction

While HTTP may be a stateless protocol, if the context it's to convey were stateless, the internet would be a much less interesting place. Many modern web architectures employ stateless application tiers, storing state in shared memory or databases. However, this is not the reality for all. Session state is immensely valuable and vast in interactive applications. This state may be stored locally for a number of reasons; for example, in applications where the data being worked is so large that network overhead is too expensive in performance. When state is stored locally to an application server, it is extremely important to the user experience that the subsequent requests continue to be delivered to the same server. Another portion of the problem is that servers should not be released until the session has finished. Working with stateful applications at scale requires an intelligent load balancer. NGINX Plus offers multiple ways to solve this problem by tracking cookies or routing. NGINX Plus's sticky directive alleviates difficulties of server affinity at the traffic controller, allowing the application to focus on its core. NGINX tracks session persistence in three ways: by creating and tracking its own cookie, detecting when applications prescribe cookies, or routing based on runtime variables.

2.0 简介

尽管 HTTP 协议是无状态的协议，但如果互联网访问都是基于无状态的访问的话，这个世界将会索然无味。但现代的 Web 架构大多基于一个无状态的应用架构，而

将会话(session)存储到内存或数据库中。可惜，这并非全部的事实。会话对于应用的交互来说功能显著，作用明显。在设计是可能出于一系列的因素考虑，将会话存储到应用服务器中；其中之一就是需要存储的会话数据太多，导致网络开销太大，将会话存储在本地服务器，能显著提升能将用户请求分发到同一台服务器的用户体验；另个方面的考虑是，如果会话没有被销毁，服务器则不应被释放掉。在具有高负载的应用服务器上使用会话，需要更加智能的负载均衡解决方案。

3.0 Introduction

For a number of reasons, applications fail. It could be because of network connectivity, server failure, or application failure, to name a few. Proxies and load balancers must be smart enough to detect failure of upstream servers and stop passing traffic to them; otherwise, the client will be waiting, only to be delivered a timeout. A way to mitigate service degradation when a server fails is to have the proxy check the health of the upstream servers. NGINX offers two different types of health checks: passive, available in the open source version; as well as active, available only in NGINX Plus. Active health checks on a regular interval will make a connection or request to the upstream server and have the ability to verify that the response is correct. Passive health checks monitor the connection or responses of the upstream server as clients make the request or connection. You may want to use passive health checks to reduce the load of your upstream servers, and you may want to use active health checks to determine failure of an upstream server before a client is served a failure

3.0 简介

访问应用时，可能由于网络连接失败，Web 服务器宕机或应用程序异常等原因导致应用程序无法访问。这时，代理或负载均衡器需要提供能够智能检测被代理或被负载均衡的 Web 服务是否无法访问的能力，来确保不会请求分发到这些失效的服务器。同时，客户端会收到连接超时的响应，结束请求等待状态。

通过代理服务器向被代理服务器发送健康检测请求，来判断被代理服务器是否失效，是

一种减轻被代理服务器压力的有效方法。NGINX 服务器提供两种不同的健康检测方案：被动检测和主动检测，开源版的 NGINX 提供被动检测功能，NGINX PLUS 提供主动检测功能。主动检测的实现原理是，NGINX 代理服务向被代理服务器定时的发送连接请求，如果被代理服务器正常响应，则说明被代理服务器正常运行。被动检测的实现原理是：NGINX 服务器通过检测客户端发送的请求及被代理(被负载均衡)服务器的响应结果进行判断被代理服务器是否失效。被动检测方案，可以有效降低被代理服务器的负载压力；主动检测则能够在客户端发送请求之前，就能够剔除掉失效服务器。

3.0 Introduction

For a number of reasons, applications fail. It could be because of network connectivity, server failure, or application failure, to name a few. Proxies and load balancers must be smart enough to detect failure of upstream servers and stop passing traffic to them; otherwise, the client will be waiting, only to be delivered a timeout. A way to mitigate service degradation when a server fails is to have the proxy check the health of the upstream servers. NGINX offers two different types of health checks: passive, available in the open source version; as well as active, available only in NGINX Plus. Active health checks on a regular interval will make a connection or request to the upstream server and have the ability to verify that the response is correct. Passive health checks monitor the connection or responses of the upstream server as clients make the request or connection. You may want to use passive health checks to reduce the load of your upstream servers, and you may want to use active health checks to determine failure of an upstream server before a client is served a failure

3.0 简介

访问应用时，可能由于网络连接失败，Web 服务器宕机或应用程序异常等原因导致应用程序无法访问。这时，代理或负载均衡器需要提供能够智能检测被代理或被负载的 Web 服务是否无法访问的能力，来确保不会请求分发到这些失效的服务器。同时，客户端会收到连接超时的响应，结束请求等待状态。

通过代理服务器向被代理服务器发送健康检测请求，来判断被代理服务器是否失效，是

一种减轻被代理服务器压力的有效方法。NGINX 服务器提供两种不同的健康检测方案：被动检测和主动检测，开源版的 NGINX 提供被动检测功能，NGINX PLUS 提供主动检测功能。主动检测的实现原理是，NGINX 代理服务向被代理服务器定时的发送连接请求，如果被代理服务器正常响应，则说明被代理服务器正常运行。被动检测的实现原理是：NGINX 服务器通过检测客户端发送的请求及被代理(被负载均衡)服务器的响应结果进行判断被代理服务器是否失效。被动检测方案，可以有效降低被代理服务器的负载压力；主动检测则能够在客户端发送请求之前，就能够剔除掉失效服务器。

3.1 What to Check

Problem

You need to check your application for health but don't know what to check.

问题

你想对服务器进行有效检测，但不止如何去检测服务器健康状况。

Solution

Use a simple but direct indication of the application health. For example, a handler that simply returns an HTTP 200 response tells the load balancer that the application process is running.

解决方案

使用一个简单粗暴的检测方案实现应用健康检测。如，负载均衡器通过获取被负载服务器的响应状态码是否为 200 判断应用服务器进程是否正常。

Discussion

It's important to check the core of the service you're load balancing for. A single comprehensive health check that ensures all of the systems are available can be problematic. Health checks should check that the application directly behind the load balancer is available over the network and that the application itself is running. With application-aware health checks, you want to pick an endpoint that simply ensures that the processes on that machine are running. It

may be tempting to make sure that the database connection strings are correct or that the application can contact its resources. However, this can cause a cascading effect if any particular service fails.

结论

实际项目中，对被负载的提供核心功能的应用服务器进行健康检测非常重要。

仅仅通过一种健康检测方案，确保核心服务是否可用，通常并不完全可靠。

健康检测应该通过网络直接检测被负载的应用服务器和应用本身是否运行正常，

来确保服务可用，这比仅使用负载均衡器来检测服务是否可用要可靠。

一般，可以选择一个功能来进行健康检测，来确保整个服务是否可用。比如，

确认数据库连接是否正常或应用是否能够正常获取它的资源。任何一个服务失效，

都可能引发蝴蝶效应导致整个服务不可用。

3.3 TCP Health Checks

Problem

You need to check your upstream TCP server for health and remove unhealthy servers from the pool.

问题

需要检测 TCP 服务器是否正常并从代理池中移除失效服务器。

Solution

Use the `health_check` directive in the server block for an active health check:

```
stream {
    server {
        listen 3306;
        proxy_pass read_backend;
        health_check interval=10 passes=2 fails=3;
    }
}
```

The example monitors the upstream servers actively. The upstream server will be considered unhealthy if it fails to respond to three or more TCP connections initiated by NGINX. NGINX performs the check every 10 seconds. The server will only be considered healthy after passing two health checks.

解决方案

在 `server` 块级指令中使用 `health_check` 简单指令，对被代理服务器进行健康检测:

```
stream {
    server {
        listen 3306;
        proxy_pass read_backend;
        health_check interval=10 passes=2 fails=3;
    }
}
```

上面的配置会对代理池中的服务器进行主动监测。如果被代理服务器未能正常

响应 NGINX 服务器的 3 个以上 TCP 连接请求，则被认为是失效的服务。

之后，NGINX 服务器会每隔 10 秒进行一次健康检测。

Discussion

TCP health can be verified by NGINX Plus either passively or actively. Passive health monitoring is done by noting the communication between the client and the upstream server. If the upstream server is timing out or rejecting connections, a passive health check will deem that server unhealthy. Active health checks will initiate their own configurable checks to determine health. Active health checks not only test a connection to the upstream server, but can expect a given response.

结论

在 NGINX PLUS 版本中同时提供被动检测和主动检测功能。被动检测是通过加之于客户端与被代理服务器的请求响应检测实现的。如果一个请求超时或者连接失败，被动检测则认为该被代理服务器失效。主动检测则是通过明确的 NGINX 指令配置来检测服务器是否失效。主动检测途径可以是一个测试的连接，也可以是一个预期的响应。

资料(译者补充)

这篇文章「[TCP Health Checks](#)」是 NGINX 服务器官网的管理员运维教程，主要讲解开源版本

和 NGINX PLUS(商业版) 的 TCP 健康检测配置处理

TODO 翻译「[TCP Health Checks](#)」

3.4 HTTP Health Checks

Problem

You need to actively check your upstream HTTP servers for health.

问题

需要主动检测 HTTP 服务器健康状态

Solution

Use the `health_check` directive in a location block:

```
http {
    server {
        ...
        location / {
            proxy_pass http://backend;
            health_check interval=2s
                        fails=2
                        passes=5
                        uri=/
                        match=welcome;
        }
    }
}

# status is 200, content type is "text/html",
# and body contains "Welcome to nginx!"
match welcome {
    status 200;
    header Content-Type = text/html;
    body ~ "Welcome to nginx!";
}
}
```

This health check configuration for HTTP servers checks the health of the upstream servers by making an HTTP request to the URI `/` every two seconds. The upstream servers must pass five consecutive health checks to be considered healthy and will be considered

unhealthy if they fail two consecutive checks. The response from the upstream server must match the defined match block, which defines the status code as 200, the header Content-Type value as 'text/html', and the string "Welcome to nginx!" in the response body.

解决方案

在 location 块级指令中使用 health_check 指令检测：

```
http {
    server {
        ...
        location / {
            proxy_pass http://backend;
            health_check interval=2s
                        fails=2
                        passes=5
                        uri=/
                        match=welcome;
        }
    }

    # status is 200, content type is "text/html",
    # and body contains "Welcome to nginx!"
    match welcome {
        status 200;
        header Content-Type = text/html;
        body ~ "Welcome to nginx!";
    }
}
```

上例，通过向被代理服务器每隔 2 秒，发送一个到 '/' URI 的请求来检测被代理服务器是否失效。被代理服务器连续接收 5 个请求，如果其中有 2 个连续请求响应失败，将被视作服务器失效。被代理服务器的健康响应格式在 match 块级指令中配置，规定响应状态码为 200, 响应 Content-Type 类型为 'text/html', 响应 body 为 "Welcome to nginx!" 字符串的响应为有效服务器。

Discussion

HTTP health checks in NGINX Plus can measure more than just

the response code. In NGINX Plus, active HTTP health checks monitor based on a number of acceptance criteria of the response from the upstream server. Active health check monitoring can be configured for how often upstream servers are checked, the URI to check, how many times it must pass this check to be considered healthy, how many times it can fail before being deemed unhealthy, and what the expected result should be. The match parameter points to a match block that defines the acceptance criteria for the response. The match block has three directives: status, header, and body. All three of these directives have comparison flags as well.

结论

在 NGINX PLUS 版本中，除了通过响应状态码来判断被代理服务器是否有效。还能够通过其它的一些响应指标来判断是否有效。如：主动检测的时间间隔(频率)，主动请求的 URI 地址，健康检测的请求次数及失败次数和预期响应结果等。在 `health_check` 指令中的 `match` 参数指向 `match` 块级指令配置，`match` 块级指令配置定义了标准的响应，包括 `status`、`header` 和 `body` 指令，他们都有各自的检测标准。

资料(译者补充)

这篇文章「[HTTP Health Checks](#)」是 NGINX 服务器官网的管理员运维教程，主要讲解开源版本

和 NGINX PLUS(商业版)的 HTTP 健康检测配置处理

TODO 翻译「[HTTP Health Checks](#)」

5.0 Introduction

Caching accelerates content serving by storing request responses to be served again in the future. Content caching reduces load to upstream servers, caching the full response rather than running computations and queries again for the same request. Caching increases performance and reduces load, meaning you can serve faster with fewer resources. Scaling and distributing caching servers in strategic locations can have a dramatic effect on user experience. It's optimal to host content close to the consumer for the best performance. You can also cache your content close to your users. This is the pattern of content delivery networks, or CDNs. With NGINX you're able to cache your content wherever you can place an NGINX server, effectively enabling you to create your own CDN. With NGINX caching, you're also able to passively cache and serve cached responses in the event of an upstream failure.

介绍

通过对请求的响应结果进行缓存，能够为后续相同请求提供加速服务。对相同请求响应内容进行内容缓存(Content Caching)，相比每次请求都重新计算和查询被代理服务器，能有效降低被代理服务器负载。内容缓存能提升服务性能，降低服务器负载压力，同时意味着能够使用更少的资源提供更快服务。可伸缩的缓存服务从架构层面来讲，能够显著提升用户体验，因为响应内容经过更少的转发就能够发送给用户，同时能提升服务器性能。

5.0 Introduction

Caching accelerates content serving by storing request responses to be served again in the future. Content caching reduces load to upstream servers, caching the full response rather than running computations and queries again for the same request. Caching increases performance and reduces load, meaning you can serve faster with fewer resources. Scaling and distributing caching servers in strategic locations can have a dramatic effect on user experience. It's optimal to host content close to the consumer for the best performance. You can also cache your content close to your users. This is the pattern of content delivery networks, or CDNs. With NGINX you're able to cache your content wherever you can place an NGINX server, effectively enabling you to create your own CDN. With NGINX caching, you're also able to passively cache and serve cached responses in the event of an upstream failure.

介绍

通过对请求的响应结果进行缓存，能够为后续相同请求提供加速服务。对相同请求响应内容进行内容缓存(Content Caching)，相比每次请求都重新计算和查询被代理服务器，能有效降低被代理服务器负载。内容缓存能提升服务性能，降低服务器负载压力，同时意味着能够使用更少的资源提供更快的服务。可伸缩的缓存服务从架构层面来讲，能够显著提升用户体验，因为响应内容经过更少的转发就能够发送给用户，同时能提升服务器性能。

5.1 Caching Zones

Problem

You need to cache content and need to define where the cache is stored.

问题

需要定义响应内容的缓存路径及缓存操作

Solution

Use the `proxy_cache_path` directive to define shared memory cache zones and a location for the content:

```
proxy_cache_path /var/nginx/cache
                  keys\_zone=CACHE:60m

                  levels=1:2

                  inactive=3h

                  max\_size=20g;
proxy_cache CACHE;
```

The cache definition example creates a directory for cached responses on the filesystem at `/var/nginx/cache` and creates a shared memory space named `CACHE` with 60 megabytes of memory. This example sets the directory structure levels, defines the release of cached responses after they have not been requested in 3 hours, and defines a maximum size of the cache of 20 gigabytes. The `proxy_cache` directive informs a particular context to use the cache zone. The `proxy_cache_path` is valid in the HTTP context, and the

`proxy_cache` directive is valid in the HTTP, server, and location contexts.

解决方案

使用 `proxy_cache_path` 指令为待缓存定义内容缓存区域的共享内存及缓存路径：

```
proxy_cache_path /var/nginx/cache
                  keys_zone=CACHE:60m

                  levels=1:2

                  inactive=3h

                  max_size=20g;
proxy_cache CACHE;
```

上面的配置中在 `proxy_cache_path` 指令中为响应在文件系统中定义了缓存的存储目录 `/var/nginx/cache`，并使用 `keys_zone` 参数创建名为 `CACHE` 的拥有 60 M 的缓存内存空间；同时通过 `levels` 参数定义目录解构级别，通过 `inactive` 参数指明如果相同请求的缓存在 3 小时内未被再次访问则被释放，并使用 `max_size` 定义了缓存最大可用存储空间为 20 G。

Discussion

To configure caching in NGINX, it's necessary to declare a path and zone to be used. A cache zone in NGINX is created with the directive `proxy_cache_path`. The `proxy_cache_path` designates a location to store the cached information and a shared memory space to store active keys and response metadata. Optional parameters to this directive provide more control over how the cache is maintained and accessed. The `levels` parameter defines how the file structure is created. The value is a colon-separated value that declares the length of subdirectory names, with a maximum of three levels. NGINX

caches based on the cache key, which is a hashed value. NGINX then stores the result in the file structure provided, using the cache key as a file path and breaking up directories based on the levels value.

The inactive parameter allows for control over the length of time a cache item will be hosted after its last use. The size of the cache is also configurable with use of the max_size parameter. Other parameters are in relation to the cache loading process, which loads the cache keys into the shared memory zone from the files cached on disk.

结论

要使用 NGINX 内容缓存，需要在配置中定义缓存目录及缓存区域(zone)。通过 proxy_cache_path 指令创建 NGINX 内容缓存，定义用于缓存信息的路径和用于存储缓存的元数据(metadata)和运行时键名(active keys)的共享内存。其它的可选参数，还提供缓存如何维护和访问的控制，levels 参数定义如何创建文件结构，定义子目录的文件名长度，语法是以冒号分隔的值，支持最大 3 级。

NGINX 的所有缓存依赖于最终被计算成散列的 cache key，接着将结果以 cache key 作为文件名，依据缓存级别创建缓存目录。

inactive 参数用于控制最后一次使用缓存选项的时间，超过这个时间的缓存会被释放。缓存的大小则可以通过 max_size 参数进行配置。还有部分参数作用于缓存加载进程中，功能是将 cache keys 从磁盘文件加载仅共享内存里。

5.2 Caching Hash Keys 配置缓存哈希键名

Problem

You need to control how your content is cached and looked up.

问题

自定义如何缓存和查找缓存内容

Solution

Use the `proxy_cache_key` directive, along with variables to define what constitutes a cache hit or miss:

```
proxy_cache_key "$host$request_uri $cookie_user";
```

This cache hash key will instruct NGINX to cache pages based on the host and URI being requested, as well as a cookie that defines the user. With this you can cache dynamic pages without serving content that was generated for a different user.

解决方案

通过一条单独的 `proxy_cache_key` 指令，以变量名的形式定义缓存命中和丢弃的规则。

```
proxy_cache_key "$host$request_uri $cookie_user";
```

上例指令依据请求域名、请求 URI 和用户 cookie 作为缓存键名，来构建 NGINX 的缓存页面。这样，就可以对动态页面进行缓存，而无需对每个用户都进行缓存内容的生成处理。

Discussion

The default `proxy_cache_key` is `"$scheme$proxy_host $request_uri"`. This default will fit most use cases. The variables used include the scheme, HTTP or HTTPS, the `proxy_host`, where the request is being sent, and the request URI. All together, this reflects the URL that NGINX is proxying the request to. You may find that there are many other factors that define a unique request per application, such as request arguments, headers, session identifiers, and so on, to which you'll want to create your own hash key. Selecting a good hash key is very important and should be thought through with understanding of the application. Selecting a cache key for static content is typically pretty straightforward; using the host-name and URI will suffice. Selecting a cache key for fairly dynamic content like pages for a dashboard application requires more knowledge around how users interact with the application and the degree of variance between user experiences. For security concerns you may not want to present cached data from one user to another without fully understanding the context. The `proxy_cache_key` directive configures the string to be hashed for the cache key. The `proxy_cache_key` can be set in the context of HTTP, server, and location blocks, providing flexible control on how requests are cached.

结论

`proxy_cache_key` 默认设置是 `"$scheme$proxy_host $request_uri"`。默认设置适用于多数的使用场景。配置之中包括 `scheme`、HTTP 或 HTTPS、代理域名 (`proxy_host`)、请求的 URI 等变量。总之，它们能够正确处理 NGINX 代理请求。

您可能会发现，对于每个应用程序，有许多其他的因素可以定义一个惟一的请求，比如请求参数、头文件、会话标识符等等，您需要创建自己的散列键。或许您已经发现，对于一个应用，还有其它的数据能够确定一个唯一的请求，比如请求参数、请求头(headers)、会话标识(session identifiers) 等等，这些都可以用于构建自己的散列键名。在构建时应基于应用程序的理解，创建选择一个好的散列键名，这一点非常重要。比较简单的是为静态内容创建缓存键名，通常，可以直接使用域名(hostname)和请求 URI 就可以了。而类似于仪表盘这类的，具有动态内容的页面，则需要充分了解用户和应用之间的交互、以及用户体验之间的差异，来构建缓存键名。如从安全的角度触发，你可能不希望缓存将一个用户的缓存数据展示给另外的用户。`proxy_cache_key` 令配置了用于缓存生成哈希值字符，此条指令可以在 HTTP、server、location 块级指令上下文中定义，实现对请求如何缓存的灵活控制。

5.3 Cache Bypass 绕过缓存

Problem

You need the ability to bypass the caching.

问题

将一些内容不进行缓存

Solution

Use the `proxy_cache_bypass` directive with a nonempty or nonzero value. One way to do this is by setting a variable within location blocks that you do not want cached to equal 1:

```
proxy_cache_bypass $http_cache_bypass;
```

The configuration tells NGINX to bypass the cache if the HTTP request header named `cache_bypass` is set to any value that is not 0.

解决方案

将 `proxy_cache_passby` 指令，设置称非空值或非 0。一种途径是，在 location 块级指令中设置一个值等于 1 的 `proxy_cache_passby` 指令：

```
proxy_cache_bypass $http_cache_bypass;
```

配置告知 NGINX 服务器，如果一个 HTTP `cache_passby` 请求头的值设置为非 0(或非空)，则不对该请求进行缓存处理。

Discussion

There are many scenarios that demand that the request is not

cached. For this, NGINX exposes a `proxy_cache_bypass` directive that when the value is nonempty or nonzero, the request will be sent to an upstream server rather than be pulled from cache. Interesting techniques and solutions for cache bypass are derived from the need of the client and application. These can be as simple as a request variable or as intricate as a number of map blocks.

For many reasons, you may want to bypass the cache. One important reason is troubleshooting and debugging. Reproducing issues can be hard if you're consistently pulling cached pages or if your cache key is specific to a user identifier. Having the ability to bypass the cache is vital. Options include but are not limited to bypassing cache when a particular cookie, header, or request argument is set. You can also turn off cache completely for a given context such as a location block by setting `proxy_cache off`;

结论

挺多应用场景下都不应对请求进行缓存处理，对此，NGINX 提供 `proxy_cache_passby` 指令来应对这些场景。通过将指令值设置为非空或非零，匹配的请求 URI 会直接发送给被代理服务器，而不是从缓存中获取。如何使用该指令，需要结合客户端和应用的实际使用。它既可以配制成如同一个请求变量一样简单，也可以配置成复杂的映射指令块。但最终目的都是绕过缓存。其中，一个重要的应用场景就是排除故障和调试应用。如果在研发过程中一直使用缓存，或对特定用户进行缓存，缓存会影响问题的复现。提供对指定 cookie、请求头(headers)或请求参数等的缓存绕过能力，则是一个必要的功能。此外，NGINX 服务器还能够在 location 块指令中将 `proxy_cache` 指令设置为 `off`，完全禁用缓存。

5.4 Cache Performance 缓存性能

Problem

You need to increase performance by caching on the client side.

问题

需要在客户端提升服务性能

Solution

Use client-side cache control headers:

```
location ~* \.(css|js)$ {
    expires 1y;

    add_header Cache-Control "public";
}
```

This location block specifies that the client can cache the content of

CSS and JavaScript files. The expires directive instructs the client

that their cached resource will no longer be valid after one year. The

add_header directive adds the HTTP response header CacheControl to the response, with a value of public, which allows any

caching server along the way to cache the resource. If we specify pri-

vate, only the client is allowed to cache the value.

解决方案

使用客户端缓存控制消息头：

```
location ~* \.(css|js)$ {
    expires 1y;

    add_header Cache-Control "public";
}
```

该 `location` 块指令设置成功后，客户端可以对 CSS 和 JS 文件进行缓存。`expires` 指令将所有缓存的有效期设置为 1 年。`add_header` 指令将 HTTP 消息头 `Cache-Control` 设置成 `public` 并加入响应中，表示所有的缓存服务器都可以缓存资源。如果将它的值设置为 `private`，则表示仅允许客户端对资源进行缓存。

Discussion

Cache performance has to do with many variables, disk speed being high on the list. There are many things within the NGINX configuration you can do to assist with cache performance. One option is to set headers of the response in such a way that the client actually caches the response and does not make the request to NGINX at all, but simply serves it from its own cache.

结论

缓存的性能和许多因素有关，其中磁盘读写速度是影响缓存性能的重要原因之一。在 NGINX 配置指令中，还有很多能够提升性能的指令。像上例中配置的，通过设置 `Cache-Control` 响应消息头，客户端会直接从本地读取缓存，而不会将请求发送给服务器来提升性能。

9.0 Introduction

User Datagram Protocol (UDP) is used in many contexts, such as DNS, NTP, and Voice over IP. NGINX can load balance over upstream servers with all the load-balancing algorithms provided to the other protocols. In this chapter, we'll cover the UDP load balancing in NGINX.

9.0 介绍

用户数据报协议(UDP)在多种场景下运用，如 DNS、NTP 服务、IP语音(Voice over IP)服务。NGINX 可以在 upstream 块级指令中使用所有的负载均衡算法实现 UDP 的负载均衡，本章将学习 UDP 负载均衡相关配置。

9.0 Introduction

User Datagram Protocol (UDP) is used in many contexts, such as DNS, NTP, and Voice over IP. NGINX can load balance over upstream servers with all the load-balancing algorithms provided to the other protocols. In this chapter, we'll cover the UDP load balancing in NGINX.

9.0 介绍

用户数据报协议(UDP)在多种场景下运用，如 DNS、NTP 服务、IP语音(Voice over IP)服务。NGINX 可以在 upstream 块级指令中使用所有的负载均衡算法实现 UDP 的负载均衡，本章将学习 UDP 负载均衡相关配置。

9.1 Stream Context

Problem

You need to distribute load between two or more UDP servers.

问题

需要在多台 UDP 服务器间实现负载均衡

Solution

Use NGINX's stream module to load balance over UDP servers

using the upstream block defined as udp:

```
stream {
    upstream ntp {

        server ntp1.example.com:123 weight=2;

        server ntp2.example.com:123;

    }

    server {

        listen 123 udp;

        proxy_pass ntp;

    }
}
```

This section of configuration balances load between two upstream NTP servers using the UDP protocol. Specifying UDP load balancing is as simple as using the `udp` parameter on the `listen` directive.

解决方案

NGINX stream 模块实现 UDP 服务器的负载均衡，作为 UDP 服务器的代理的

upstream 块级指令被定义称使用 **UDP** 协议:

```
stream {
    upstream ntp {

        server ntp1.example.com:123 weight=2;

        server ntp2.example.com:123;

    }

    server {

        listen 123 udp;

        proxy_pass ntp;

    }
}
```

示例中，对 2 台使用 **UDP** 协议的 **NTP** 服务器进行负载均衡代理。实现 **UDP** 协议的负载均衡，简单到仅需在 **server** 指令块中的 **listen** 指令加上一个 **udp** 参数就可以了。

Discussion

One might ask, “Why do you need a load balancer when you can have multiple hosts in a DNS A or SRV record?” The answer is that not only are there alternative balancing algorithms we can balance with, but we can load balance over the DNS servers themselves. UDP services make up a lot of the services that we depend on in networked systems such as DNS, NTP, and Voice over IP. UDP load balancing may be less common to some but just as useful in the world of scale.

UDP load balancing will be found in the stream module, just like TCP, and configured mostly in the same way. The main difference is that the listen directive specifies that the open socket is for work-

ing with datagrams. When working with datagrams, there are some other directives that may apply where they would not in TCP, such as the `proxy_response` directive that tells NGINX how many expected responses may be sent from the upstream server, by default being unlimited until the `proxy_timeout` limit is reached.

结论

或许有人会问“既然有多条 A 记录或 SRV 记录的 DNS 域名解析，为什么我还需要使用 NGINX 的负载均衡功能呢？”我们的理由是，NGINX 不仅提供了多种负载均衡算法，而且还能对 DNS 服务器本身进行负载均衡处理。UDP 协议构建了 DNS 解析、NTP 服务器、IP 语音服务等大量基础服务。UDP 负载均衡在某些场景下运用不是特别广泛，但在整个网络世界则并非如此。

UDP 负载均衡同 TCP 负载均衡一样集成在 `stream` 模块内，并且它们的使用方法也几乎一样。二者的主要区别是，在 `listen` 指令中定义用于 UDP 协议的套接字及 `udp` 参数。此外，还有一些仅用于 UDP 协议的指令，像 `proxy_response` 指令，`proxy_response` 指令告知 NGINX 服务器从被代理服务器接收多少预期响应，默认是无限制的，直到达到 `proxy_timeout` 设定值。

9.3 Health Checks

Problem

You need to check the health of upstream UDP servers.

问题

检测 upstream 指令中的 UDP 服务器是否健康。

Solution

Use NGINX health checks with UDP load balancing to ensure only healthy upstream servers are sent datagrams:

```
upstream ntp {  
    server ntp1.example.com:123 max\_fails=3 fail\_timeout=3s;  
    server ntp2.example.com:123 max\_fails=3 fail\_timeout=3s;  
}
```

This configuration passively monitors the upstream health, setting the max_fails directive to 3, and fail_timeout to 3 seconds.

解决方案

对 UDP 负载均衡配置进行健康检测，确保只对正常运行的 UDP 服务器发送

数据报文：

```
upstream ntp {  
    server ntp1.example.com:123 max\_fails=3 fail\_timeout=3s;  
    server ntp2.example.com:123 max\_fails=3 fail\_timeout=3s;  
}
```

配置采用被动检测功能，将 max_fails 指令设置为 3 次，fail_timeout 设置为 3 秒。

Discussion

Health checking is important on all types of load balancing not only from a user experience standpoint but also for business continuity. NGINX can actively and passively monitor upstream UDP servers to ensure they're healthy and performing. Passive monitoring watches for failed or timed-out connections as they pass through NGINX. Active health checks send a packet to the specified port, and can optionally expect a response.

结论

无论何种负载均衡，无论从用户体验角度，还是商业角度，健康检测都至关重要。NGINX 同样提供 UDP 负载均衡主动和被动检测方案。被动检测会监测连接失败及超时请求作为失效服务判断。主动检测会主动发送数据包值指定端口，通过配置的预期响应判断服务是否有效。

Part II: Security and Access

This is Part II of III of NGINX Cookbook. This part will focus on security aspects and features of NGINX and NGINX Plus, the licensed version of the NGINX server. Throughout this part, you will learn the basics about controlling access and limiting abuse and misuse of your web assets and applications. Security concepts such as encryption of your web traffic as well as basic HTTP authentication will be explained as applicable to the NGINX server. More advanced topics are covered as well, such as setting up NGINX to verify authentication via third-party systems as well as through JSON Web Token Signature validation and integrating with Single sign-on providers. This part covers some amazing features of NGINX and NGINX Plus such as securing links for time-limited access and security as well as enabling Web Application Firewall capabilities of NGINX Plus with the ModSecurity module. Some of the plug-and-play modules in this part are only available through the paid NGINX Plus subscription, however this does not mean that the core open source NGINX server is not capable of these securities.

Part II: Security and Access

This is Part II of III of NGINX Cookbook. This part will focus on security aspects and features of NGINX and NGINX Plus, the licensed version of the NGINX server. Throughout this part, you will learn the basics about controlling access and limiting abuse and misuse of your web assets and applications. Security concepts such as encryption of your web traffic as well as basic HTTP authentication will be explained as applicable to the NGINX server. More advanced topics are covered as well, such as setting up NGINX to verify authentication via third-party systems as well as through JSON Web Token Signature validation and integrating with Single sign-on providers. This part covers some amazing features of NGINX and NGINX Plus such as securing links for time-limited access and security as well as enabling Web Application Firewall capabilities of NGINX Plus with the ModSecurity module. Some of the plug-and-play modules in this part are only available through the paid NGINX Plus subscription, however this does not mean that the core open source NGINX server is not capable of these securities.

本书的第二部分将讲解 NGINX 和 NGINX PLUS 版本的安全特性。通过第二部分相关知识，您将掌握如何配置 NGINX 服务器才能有效控制服务器资源不被应用程序滥用。学习安全配置，如 NGINX 服务器如何使用对请求数据加密和基本的 HTTP 认证。更高级的安全配置，像 NGINX 服务器如何使用第三方认证系统进行身份认证，如何使用 JSON 令牌校验和单点登录功能等。此外，您还将学习 NGINX 和 NGINX PLUS 版本更多惊艳的特性，如访问次数控制、使用 NGINX PLUS 版本的 ModSecurity 模块开启防火墙功能等等。对于一些即插即用(plug-and-pay)模块，仅能通过 NGINX PLUS 版本订阅获取，然而，这并不意

意味着免费版的 NGINX 服务器不能使用。

CHAPTER 11 Controlling Access 访问控制

11.0 Introduction

Controlling access to your web applications or subsets of your web applications is important business. Access control takes many forms in NGINX, such as denying it at the network level, allowing it based on authentication mechanisms, or HTTP responses instructing browsers how to act. In this chapter we will discuss access control based on network attributes, authentication, and how to specify Cross-Origin Resource Sharing (CORS) rules.

11.0 介绍

项目中实现对 web 应用程序或 web 应用程序子系统的访问控制是项目的重要组成部分。实现 NGINX 的访问控制形式多样，比如从网络层面实现访问控制，允许 NGINX 采用身份校验机制，或通过 HTTP 响应引导浏览器如何操作。本章将讨论使用网络属性(network attributes)、身份认证、跨域资源共享(CORS : Cross-Origin Resource Sharing)原则等安全控制知识。

11.1 Access Based on IP Address | 基于 IP 地址访问配置

Problem

You need to control access based on the IP address of the client.

问题

需要基于客户端的 IP 地址实现访问控制功能

Solution

Use the HTTP access module to control access to protected resources:

```
location /admin/ {  
    deny 10.0.0.1;  
    allow 10.0.0.0/20;  
    allow 2001:0db8::/32;  
    deny all;  
}
```

The given location block allows access from any IPv4 address in 10.0.0.0/20 except 10.0.0.1, allows access from IPv6 addresses in the 2001:0db8::/32 subnet, and returns a 403 for requests originating from any other address. The allow and deny directives are valid within the HTTP, server, and location contexts. Rules are checked in sequence until a match is found for the remote address.

解决方案

使用 HTTP 的 access 模块，实现对受保护资源的访问控制：


```
location /admin/ {  
    deny 10.0.0.1;  
    allow 10.0.0.0/20;  
    allow 2001:0db8::/32;  
    deny all;  
}
```

给定的 `location` 块级指令中配置了允许除 10.0.0.1 外的所有 10.0.0.0/20 IPv4 地址访问，同时允许 2001:0db8::/32 及其子网的 IPv6 地址访问，其它 IP 地址的访问将会收到 HTTP 状态为 403 的响应。`allow` 和 `deny` 指令可在 HTTP、`server`、`location` 上下文中使用。控制规则依据配置的顺序进行查找，直到匹配到控制规则。

Discussion

Protecting valuable resources and services on the internet must be done in layers. NGINX provides the ability to be one of those layers. The `deny` directive blocks access to a given context, while the `allow` directive can be used to allow subsets of the blocked access. You can use IP addresses, IPv4 or IPv6, CIDR block ranges, the keyword `all`, and a Unix socket. Typically when protecting a resource, one might allow a block of internal IP addresses and deny access from all.

结论

需要控制访问的资源需要实现分层控制。NGINX 服务器提供对资源进行分层控制的能力。`deny` 指令会限制对给定上下文的访问，`allow` 指令与 `deny` 功能相反，它们的值可以是定值 IP 地址、IPv4 或 IPv6 地址、无类别域间路由(CIDR: Classless Inter-Domain Routing)、关键字或 UNIX 套接字。IP 限制的常用解决方案是，允许一个内部的 IP 地址访问资源，拒绝其它所有 IP 地址的访问来实现对资源的访问控制。

11.2 Allowing Cross-Origin Resource Sharing | 跨域资源共享控制

Problem

You're serving resources from another domain and need to allow CORS to enable browsers to utilize these resources.

问题

项目资源部署在其它域名，允许跨域的访问请求使用这些资源。

Solution

Alter headers based on the request method to enable CORS:

解决方案

通过对不同请求方法设置对应的 HTTP 消息头实现跨域资源共享：

```
map $request_method $cors_method {
    OPTIONS 11;
    GET 1;
    POST 1;
    default 0;
}

server {
    ...

    location / {
        if \($cors\_method ~ '1'\) {
            add\_header 'Access-Control-Allow-Methods' 'GET,POST,OPTIONS';
            add\_header 'Access-Control-Allow-Origin' '\*.example.com';
            add\_header 'Access-Control-Allow-Headers'
                'DNT,
                Keep-Alive,
                User-Agent,
                X-Requested-With,
                If-Modified-Since,
                Cache-Control,
                Content-Type';
        }

        if \($cors\_method = '11'\) {
            add\_header 'Access-Control-Max-Age' 1728000;
            add\_header 'Content-Type' 'text/plain; charset=UTF-8';
            add\_header 'Content-Length' 0;
            return 204;
        }
    }
}
```

There's a lot going on in this example, which has been condensed by using a map to group the GET and POST methods together. The OPTIONS request method returns information called a preflight request to the client about this server's CORS rules. OPTIONS, GET, and POST methods are allowed under CORS. Setting the AccessControl-Allow-Origin header allows for content being served from this server to also be used on pages of origins that match this header. The preflight request can be cached on the client for 1,728,000 seconds, or 20 days.

这个示例包含很多内容，首先使用 `map` 指令将 `GET` 和 `POST` 请求分入同一组。

The `OPTIONS` request method returns information called a preflight request to the client about this server's CORS rules。在配置中 `GET`、`POST`、`OPTIONS` 请求都允许跨域访问资源。`Access-Controll-Allow-Origin` 消息头设置允许请求服务器资源的域名，当客户端域名匹配该设置的域名规则时，则可以访问服务器资源。The preflight request can be cached on the client for 1,728,000 seconds, or 20 days.

Discussion

Resources such as JavaScript make cross-origin resource requests when the resource they're requesting is of a domain other than its own origin. When a request is considered cross origin, the browser is required to obey CORS rules. The browser will not use the resource if it does not have headers that specifically allow its use. To allow our resources to be used by other subdomains, we have to set the CORS headers, which can be done with the `add_header` directive. If the request is a `GET`, `HEAD`, or `POST` with standard content type, and the request does not have special headers, the browser will make the request and only check for origin. Other request methods will cause the browser to make the preflight request to check the terms of the server to which it will obey for that resource. If you do not set these headers appropriately, the browser will give an error when trying to utilize that resource.

结论

当请求的资源不属于当前域名时，就会产生一个跨域的请求，比如 JavaScript 请求其它域名的资源变产生跨域请求。当跨域请求产生，浏览器则必须遵守

跨域资源共享(CORS)规则，此时浏览器便不会引用这些跨域的资源，除非，检测到给定的允许使用非同源资源的 HTTP 消息头。为满足子域名间能够跨域使用资源，我们需要使用 `add_header` 指令设置对应的 CORS 消息头。如果一个 HTTP 请求是标准的 GET、POST 或 HEAD 请求且并未设置特定的消息头，浏览器就回对请求和源域名进行检测。Other request methods will cause the browser to make the preflight request to check the terms of the server to which it will obey for that resource.如果没有实现对特定请求消息头的配置，浏览器在获取跨域资源时，则会抛出错误禁止应用跨域资源。

扩展资料

[[译]nginx map(ngx_http_map_module)](<https://www.jianshu.com/p/6dea80baba9b>)

CHAPTER 12 Limiting Use

12.0 Introduction

Limiting use or abuse of your system can be important for throttling heavy users or stopping attacks. NGINX has multiple modules built in to help control the use of your applications. This chapter focuses on limiting use and abuse, the number of connections, the rate at which requests are served, and the amount of bandwidth used. It's important to differentiate between connections and requests: connections (TCP connections) are the transport layer on which requests are made and therefore are not the same thing. A browser may open multiple connections to a server to make multiple requests. However, in HTTP/1 and HTTP/1.1, requests can only be made one at a time on a single connection; whereas in HTTP/2, multiple requests can be made in parallel over a single TCP connection. This chapter will help you restrict usage of your service and mitigate abuse.

限制控制的合理使用对于服务器来讲能够有效阻止攻击者攻击请求。NGINX 服务器内置多种限制控制模块。本章将深入讲解访问连接数、请求速率和带宽限制等功能。首先需要区分什么是连接(connections) 和 请求(requests):连接(TCP 连接)是位于传输层的协议，一个 HTTP 请求会产生一个连接，所以它们是不同的东西。浏览器与服务器之间的交互允许打开多个连接，这样可以同时发起多个请求。然而，HTTP/1 和 HTTP/1.1 协议，同一时间仅能处理一个连接，直到 HTTP/2 才突破此限制，支持同时处理多个连接。本章将学习相关限制特性，实现对服务的合理使用。

CHAPTER 12 Limiting Use

12.0 Introduction

Limiting use or abuse of your system can be important for throttling heavy users or stopping attacks. NGINX has multiple modules built in to help control the use of your applications. This chapter focuses on limiting use and abuse, the number of connections, the rate at which requests are served, and the amount of bandwidth used. It's important to differentiate between connections and requests: connections (TCP connections) are the transport layer on which requests are made and therefore are not the same thing. A browser may open multiple connections to a server to make multiple requests. However, in HTTP/1 and HTTP/1.1, requests can only be made one at a time on a single connection; whereas in HTTP/2, multiple requests can be made in parallel over a single TCP connection. This chapter will help you restrict usage of your service and mitigate abuse.

限制控制的合理使用对于服务器来讲能够有效阻止攻击者攻击请求。NGINX 服务器内置多种限制控制模块。本章将深入讲解访问连接数、请求速率和带宽限制等功能。首先需要区分什么是连接(connections) 和 请求(requests):连接(TCP 连接)是位于传输层的协议，一个 HTTP 请求会产生一个连接，所以它们是不同的东西。浏览器与服务器之间的交互允许打开多个连接，这样可以同时发起多个请求。然而，HTTP/1 和 HTTP/1.1 协议，同一时间仅能处理一个连接，直到 HTTP/2 才突破此限制，支持同时处理多个连接。本章将学习相关限制特性，实现对服务的合理使用。

12.1 Limiting Connections | 连接数限制

Problem

You need to limit the number of connections based on a predefined key, such as the client's IP address.

问题

基于给定的规则如 IP 地址，实现请求连接数。

Solution

Construct a shared memory zone to hold connection metrics, and use the `limit_conn` directive to limit open connections:

```
http {
    limit_conn_zone $binary_remote_addr zone=limitbyaddr:10m;
    limit_conn_status 429;
    ...
    server {
        ...
        limit_conn limitbyaddr 40;
        ...
    }
}
```

This configuration creates a shared memory zone named `limitbyaddr`.

The predefined key used is the client's IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The `limit_conn` directive takes two parameters: a

`limit_conn_zone` name, and the number of connections allowed.

The `limit_conn_status` sets the response when the connections are limited to a status of 429, indicating too many

requests. The `limit_conn` and `limit_conn_status` directives are valid in the HTTP, server, and location context.

解决方案

使用 `limit_conn_zone` 指令构建存储当前连接数的内存区域；然后，使用 `limit_conn` 指令设置支持的连接数：

```
http {
    limit_conn_zone $binary_remote_addr zone=limitbyaddr:10m;
    limit_conn_status 429;
    ...
    server {
        ...
        limit_conn limitbyaddr 40;
        ...
    }
}
```

配置中创建了一个名为 `limitbyaddr` 的存储容量为 10 M 的共享内存，键名则为客户端二进制的 IP 地址。`limit_conn` 指令接收两个参数：一个是 `limit_conn_zone` 创建的名称 `limitbyaddr`，和支持的连接数 40。`limit_conn_status` 指令定义了当连接数超过 40 个时的响应状态码。`limit_conn` 和 `limit_conn_status` 指令能够在 HTTP、server 和 location 上下文中使用。

Discussion

Limiting the number of connections based on a key can be used to defend against abuse and share your resources fairly across all your clients. It is important to be cautious of your predefined key. Using an IP address, as we are in the previous example, could be dangerous if many users are on the same network that originates from the same IP, such as when behind a Network Address Translation (NAT). The entire group of clients will be limited. The `limit_conn_zone`

directive is only valid in the HTTP context. You can utilize any number of variables available to NGINX within the HTTP context in order to build a string on which to limit by. Utilizing a variable that can identify the user at the application level, such as a session cookie, may be a cleaner solution depending on the use case. The `limit_conn_status` defaults to 503, service unavailable. You may find it preferable to use a 429, as the service is available, and 500-level responses indicate server error whereas 400-level responses indicate client error.

结论

合理使用连接数限制，可以是服务器的资源被各个客户端合理使用。使用的关键在于定义一个合理的存储键名。本例中基于 IP 地址作为存储键名不是一个好的选择，因为，一旦有许多用户通过同一网络访问服务，便会限制该 IP 地址的所有用户的访问连接数，这很不合理。`limit_conn_zone` 仅在 http 上下文中可用可以使用所有的 NGINX 变量来构建限制键名。通过使用能够识别用户会话的变量如 `cookie`，有利于合理使用连接控制功能。`limit_conn_status` 默认状态码是 503 服务不可用。例子中使用 429 因为服务是可用的，而 500 级的响应码表示服务器内部错误，而 400 级的响应码表示客户端错误。

2.2 Limiting Rate 限速

Problem

You need to limit the rate of requests by predefined key, such as the client's IP address.

问题

依据某些规则对用户请求进行限速，如通过用户 IP 地址进行限速。

Solution

Utilize the rate-limiting module to limit the rate of requests:

```
http {
    limit_req_zone $binary_remote_addr zone=limitbyaddr:10m rate=1r/s;
    limit_req_status 429;
    ...
    server {
        ...
        limit_req zone=limitbyaddr burst=10 nodelay;
        ...
    }
}
```

This example configuration creates a shared memory zone named `limitbyaddr`. The predefined key used is the client's IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The zone sets the rate with a keyword argument. The `limit_req` directive takes two optional keyword arguments: `zone` and `burst`. `zone` is required to instruct the directive on which shared memory request limit zone to use. When the request rate for a given zone is exceeded, requests are delayed until their maximum burst size is reached, denoted by the `burst` keyword argument. The burst

keyword argument defaults to zero. `limit_req` also takes a third optional parameter, `nodelay`. This parameter enables the client to use its burst without delay before being limited. `limit_req_status` sets the status returned to the client to a particular HTTP status code; the default is 503. `limit_req_status` and `limit_req` are valid in the context of HTTP, server, and location. `limit_req_zone` is only valid in the HTTP context.

解决方案

利用 `rate-limiting` 模块实现对请求限速：

```
http {
    limit_req_zone $binary_remote_addr zone=limitbyaddr:10m rate=1r/s;
    limit_req_status 429;
    ...
    server {
        ...
        limit_req zone=limitbyaddr burst=10 nodelay;
        ...
    }
}
```

实例中，创建了一个 10 M 存储空间名为 `limitbyaddr` 的共享内存，并使用二进制的客户端 IP 地址作为键名。`limit_req_zone` 还设置了访问速度。

`limit_req` 指令主要包含两个可选参数：`zone` 和 `burst`。`zone` 参数值即为

`limit_req_zone` 指令中 `zone` 参数定义的存储空间名。当用户请求超出限速

设置时，超出的请求将会存储至 `burst` 定义的缓冲区，直至也超出请求限速缓冲

速率，这是将响应 429 状态码给客户端。`burst` 参数默认值为 0。此外，`limit_req`

还有第三个参数 `nodelay`：它的功能是提供瞬时处理 `rate + burst` 个请求的能力。

`limit_req_status` 参数用于设置超出速率请求响应给客户端的状态码，默认是 503，

示例中设置为 429。`limit_req_status` 和 `limit_req` 指令适用于 HTTP、server 和

location 上下文。`limit_req_zone` 指令仅能在 HTTP 上下文中使用。

Discussion

The rate-limiting module is very powerful in protecting against abusive rapid requests while still providing a quality service to everyone. There are many reasons to limit rate of request, one being security. You can deny a brute force attack by putting a very strict limit on your login page. You can disable the plans of malicious users that might try to deny service to your application or to waste resources by setting a sane limit on all requests. The configuration of the rate-limit module is much like the preceding connectionlimiting module described in Recipe 12.1, and much of the same concerns apply. The rate at which requests are limited can be done in requests per second or requests per minute. When the rate limit is hit, the incident is logged. There's a directive not in the example: `limit_req_log_level`, which defaults to `error`, but can be set to `info`, `notice`, or `warn`.

结论

`rate-limiting` 模块在项目中非常有用，通过防止瞬间爆发的请求，为每个用户提供高质量的服务。使用限速模块有诸多理由，其一是处于安全方面考虑。如在登录页面设置严格的限速控制，拒绝暴力攻击。如果没有依据用户实现限速功能，可能会导致其他用户无法使用服务或浪费了服务器资源。`rate-limiting` 模块有点类似上一章节中讲解的限制连接模块。限速设置可以依据每秒限速，也可依据每分钟进行限速。当用户请求满足限速条件时，请求将被记入日志中。另外，还有一条指令没有示例中给出：`limit_req_log_level` 指令设置限速日志级别，它默认值为 `error` 级别，您还可以设置为 `info`、`notice` 或 `warn` 级别。

参考资料

[Nginx下limit_req模块burst参数超详细解析]

(http://blog.csdn.net/hellow__world/article/details/78658041)

12.3 Limiting Bandwidth 限制带宽

Problem

You need to limit download bandwidths per client for your assets.

问题

需要依据客户端，限制它们下载速度。

Solution

Utilize NGINX's `limit_rate` and `limit_rate_after` directives to

limit the rate of response to a client:

```
location /download/ {  
    limit_rate_after 10m;  
    limit_rate 1m;  
}
```

The configuration of this location block specifies that for URIs with the prefix `download`, the rate at which the response will be served to the client will be limited after 10 megabytes to a rate of 1 megabyte per second. The bandwidth limit is per connection, so you may want to institute a connection limit as well as a bandwidth limit where applicable.

解决方案

使用 NGINX 服务器的 `limit_rate` 和 `limit_rate_after` 指令实现客户端响应速度：

```
location /download/ {  
    limit_rate_after 10m;  
    limit_rate 1m;  
}
```

`location` 块级指令设置对于匹配 `/download/` 前缀的 URI 请求，当客户端下载数据达到 10 M 以后，对其下载速度限制在 1 M 以内。不过该带宽限制功能仅仅是针对单个连接而言，因而，可能实际使用中需要配合使用连接限制和带宽限制实现下载限速。

Discussion

Limiting the bandwidth for particular connections enables NGINX to share its upload bandwidth across all of the clients in a manner you specify. These two directives do it all: `limit_rate_after` and `limit_rate`. The `limit_rate_after` directive can be set in almost any context: `http`, `server`, `location`, and if when the if is within a `location`. The `limit_rate` directive is applicable in the same contexts as `limit_rate_after`; however, it can alternatively be set by setting a variable named `$limit_rate`. The `limit_rate_after` directive specifies that the connection should not be rate limited until after a specified amount of data has been transferred. The `limit_rate` directive specifies the rate limit for a given context in bytes per second by default. However, you can specify `m` for megabytes or `g` for gigabytes. Both directives default to a value of 0. The value 0 means not to limit download rates at all. This module allows you to programmatically change the rate limit of clients.

结论

`limit_rate_after` 和 `limit_rate` 使 NGINX 能够以您指定的方式在所有客户端上共享其上传带宽。`limit_rate` 和 `limit_rate_after` 指令可在几乎所有的上下文中使用，如 `http`、`server`、`location`、`location` 指令内的 `if` 指令，不过 `limit_rate` 指令还可以通过 `$limit_rate` 变量来设置带宽。`limit_rate_after` 指令表示在客户端使用多少流量后，将启用带宽限制功能。

`limit_rate` 指令默认限速单位为字节(byte)，还可以设置为 `m` (兆字节) 和 `g` (吉字节)。这两条指令的默认值都是 `0`，表示不对带宽进行任何限制。另外，该模块提供以编码方式对客户端带宽进行限速。

参考资料

[Nginx带宽控制](<https://huoding.com/2015/03/20/423/>)

13.0 Introduction

The internet can be a scary place, but it doesn't have to be. Encryption for information in transit has become easier and more attainable in that signed certificates have become less costly with the advent of Let's Encrypt and Amazon Web Services. Both offer free certificates with limited usage. With free signed certificates, there's little standing in the way of protecting sensitive information. While not all certificates are created equal, any protection is better than none.

In this chapter, we discuss how to secure information between NGINX and the client, as well as NGINX and upstream services.

13.0 介绍

虽然总体上互联网环境危机四伏，但并非没有解决方法。随着 Let's 加密和 Amazon Web 服务的出现，在传输中传输对信息加密变得更加容易，也更容易实现。这两个服务都提供免费的证书可供开发这使用。这些免费证书，使用户敏感信息保护不再成为技术障碍。然并非所有的证书在保护敏感信息的能力都是相同的，不过启用总比不用强。本章将学习客户端与 NGINX 数据加密和 NGINX 服务器与代理服务(upstream services)的数据加密解决方案。

13.0 Introduction

The internet can be a scary place, but it doesn't have to be. Encryption for information in transit has become easier and more attainable in that signed certificates have become less costly with the advent of Let's Encrypt and Amazon Web Services. Both offer free certificates with limited usage. With free signed certificates, there's little standing in the way of protecting sensitive information. While not all certificates are created equal, any protection is better than none.

In this chapter, we discuss how to secure information between NGINX and the client, as well as NGINX and upstream services.

13.0 介绍

虽然总体上互联网环境危机四伏，但并非没有解决方法。随着 Let's 加密和 Amazon Web 服务的出现，在传输中传输对信息加密变得更加容易，也更容易实现。这两个服务都提供免费的证书可供开发这使用。这些免费证书，使用户敏感信息保护不再成为技术障碍。然并非所有的证书在保护敏感信息的能力都是相同的，不过启用总比不用强。本章将学习客户端与 NGINX 数据加密和 NGINX 服务器与代理服务(upstream services)的数据加密解决方案。

13.1 Client-Side Encryption 客户端加密

Problem

You need to encrypt traffic between your NGINX server and the client.

问题

客户端与 NGINX 服务器之间的请求数据需要加密处理。

Solution

Utilize one of the SSL modules, such as the `ngx_http_ssl_module` or `ngx_stream_ssl_module` to encrypt traffic:

```
http {
    \# All directives used below are also valid in stream
    server {
        listen 8443 ssl;
        ssl\_protocols TLSv1.2;
        ssl\_ciphers HIGH:!aNULL:!MD5;
        ssl\_certificate /usr/local/nginx/conf/cert.pem;
        ssl\_certificate\_key /usr/local/nginx/conf/cert.key;
        ssl\_session\_cache shared:SSL:10m;
        ssl\_session\_timeout 10m;
    }
}
```

This configuration sets up a server to listen on a port encrypted with SSL, 8443. The server accepts the SSL protocol version TLSv1.2. The SSL certificate and key locations are disclosed to the server for use. The server is instructed to use the highest strength offered by the client while restricting a few that are insecure. The SSL session cache and timeout allow for workers to cache and store session parameters for a given amount of time. There are many other session cache

options that can help with performance or security of all types of use cases. Session cache options can be used in conjunction. However, specifying one without the default will turn off that default, built-in session cache.

解决方案

启用 `ngx_http_ssl_module` 或 `ngx_stream_ssl_module` 其中之一 NGINX SSL 模块对数据进行加密：

```
http {
    \# All directives used below are also valid in stream
    server {
        listen 8433 ssl;
        ssl_protocols TLSv1.2;
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_certificate /usr/local/nginx/conf/cert.pem;
        ssl_certificate_key /usr/local/nginx/conf/cert.key;
        ssl_session_cache shared:SSL:10m;
        ssl_session_timeout 10m;
    }
}
```

实例在 `server` 块级指令中设置监听启用 `ssl` 加密的 `8843` 端口。使用的 `ssl` 协议为 `TLS1.2` 版本。服务器有访问 `SSL` 证书及密钥目录的权限。另外，服务器和客户端交互采用最高强度加密数据。`ssl_session_cache` 和 `ssl_session_timeout` 指令用于设置会话存储内存空间和时间，除这两个指令外，还有一些与会员有关的指令，可以用于提升性能和安全性。However, specifying one without the default will turn off that default, built-in session cache.

Discussion

Secure transport layers are the most common way of encrypting information in transit. At the time of writing, the Transport Layer Security protocol (TLS) is the default over the Secure Socket Layer (SSL) protocol. That's because versions 1 through 3 of SSL are now

considered insecure. While the protocol name may be different, TLS still establishes a secure socket layer. NGINX enables your service to protect information between you and your clients, which in turn protects the client and your business. When using a signed certificate, you need to concatenate the certificate with the certificate authority chain. When you concatenate your certificate and the chain, your certificate should be above the chain in the file. If your certificate authority has provided many files in the chain, it is also able to provide the order in which they are layered. The SSL session cache enhances performance by not having to negotiate for SSL/TLS versions and ciphers.

结论

安全传输层是加密传输数据的常用手段。在写作本书时，传输层安全协议(TSL)是安全套接字层协议(SSL)的默认协议，因为，现在认为 1.0 到 3.0 版本的 SSL 协议都是不安全的。尽管安全协议的名称有所不同，但无论 TSL 协议还是 SSL 协议它们的最终目的都是构建一个安全的套接层。NGINX 服务器让你能在服务与客户端之间构建加密的数据传输，保证业务与用户数据安全。使用签名证书时，需要将证书与证书颁发机构链连接起来。证书和颁发机构通信时时，你的证书应该在文件链中。如果您的证书颁发机构在链中提供了许多文件，它也能够提供它们分层的顺序。SSL 会话缓存性能通过不带版本信息和数据加密方式的 SSL / TLS 协议实现。

Also See

[Mozilla Server Side TLS Page](#)

[Mozilla SSL Configuration Generator](#)

[Test your SSL Configuration with SSL Labs SSL Test](#)

3.2 Upstream Encryption

Problem

You need to encrypt traffic between NGINX and the upstream service and set specific negotiation rules for compliance regulations or if the upstream is outside of your secured network.

问题

需要在 NGINX 与 upstream 代理服务器之间依据具体规则构建安全通信。

Solution

Use the SSL directives of the HTTP proxy module to specify SSL rules:

```
location / {
    proxy_pass https://upstream.example.com;
    proxy_ssl_verify on;
    proxy_ssl_verify_depth 2;
    proxy_ssl_protocols TLSv1.2;
}
```

These proxy directives set specific SSL rules for NGINX to obey. The configured directives ensure that NGINX verifies that the certificate and chain on the upstream service is valid up to two certificates deep. The proxy_ssl_protocols directive specifies that NGINX will only use TLS version 1.2. By default NGINX does not verify upstream certificates and accepts all TLS versions.

解决方案

使用 http 模块的 ssl 指令构建具体的 SSL 通信规则:

```
location / {  
    proxy_pass https://upstream.example.com;  
    proxy_ssl_verify on;  
    proxy_ssl_verify_depth 2;  
    proxy_ssl_protocols TLSv1.2;  
}
```

示例中配置了 NGINX 与代理服务器之间通信的 SSL 规则。首先启用安全传输校验功能，并将 NGINX 与代理服务器之间的证书校验深度设置为 2 层。proxy_ssl_protocols 指令用于设置使用 TLS 1.2 版本协议，它的默认值是不会校验证证书，并可以使用所有版本 TLS 协议。

Discussion

The configuration directives for the HTTP proxy module are vast, and if you need to encrypt upstream traffic, you should at least turn on verification. You can proxy over HTTPS simply by changing the protocol on the value passed to the proxy_pass directive. However, this does not validate the upstream certificate. Other directives available, such as proxy_ssl_certificate and proxy_ssl_certificate_key, allow you to lock down upstream encryption for enhanced security. You can also specify proxy_ssl_crl or a certificate revocation list, which lists certificates that are no longer considered valid. These SSL proxy directives help harden your system's communication channels within your own network or across the public internet.

结论

HTTP proxy 模块的指令繁多，如果需要启用安全传输功能，至少也需要开启校验功能。此外，我们还可以对 proxy_pass 指令设置协议，来实现 HTTPS 传输。不过，这种方式不会对被代理服务器的证书进行校验。其它的指令，如 proxy_ssl_certificate 和

`proxy_ssl_certificate_key` 指令，用于配置被代理服务器待校证书目录。另外，还有 `proxy_ssl_crl` 和 无效证书列表功能，用于列出无需认证的证书。这些 proxy 模块的 SSL 指令能够助你构建安全的内部服务通信和互联网通信。

Practical Security Tips

20.0 Introduction

Security is done in layers, and much like an onion, there must be multiple layers to your security model for it to be truly hardened. In Part II of this book, we've gone through many different ways to secure your web applications with NGINX and NGINX Plus. Many of these security methods can be used in conjunction to help harden security. The following are a few more practical security tips to ensure your users are using HTTPS and to tell NGINX to satisfy one or more security methods.

20.0 介绍

我们的系统通常是分层的，所以安全策略需要依据不同的分层架构指定解决方案。在本书的第二部分，已经介绍了诸多安全策略方案。其中的部分章节中的解决方案能够用于加强安全防御能力。在这个章节，将从实战角度出发，讲解构建安全的 HTTPS 协议和 NGINX 服务器的方法。

Practical Security Tips

20.0 Introduction

Security is done in layers, and much like an onion, there must be multiple layers to your security model for it to be truly hardened. In Part II of this book, we've gone through many different ways to secure your web applications with NGINX and NGINX Plus. Many of these security methods can be used in conjunction to help harden security. The following are a few more practical security tips to ensure your users are using HTTPS and to tell NGINX to satisfy one or more security methods.

20.0 介绍

我们的系统通常是分层的，所以安全策略需要依据不同的分层架构指定解决方案。在本书的第二部分，已经介绍了诸多安全策略方案。其中的部分章节中的解决方案能够用于加强安全防御能力。在这个章节，将从实战角度出发，讲解构建安全的 HTTPS 协议和 NGINX 服务器的方法。

20.1 HTTPS Redirects 重定向至 HTTPS 协议

Problem

You need to redirect unencrypted requests to HTTPS.

问题

需要将用户请求从 HTTP 协议重定向至 HTTPS 协议。

Solution

Use a rewrite to send all HTTP traffic to HTTPS:

```
server {  
    listen 80 default_server;  
    listen [::]:80 default_server;  
    server_name _;  
    return 301 https://$host$request_uri;  
}
```

This configuration listens on port 80 as the default server for both IPv4 and IPv6 and for any hostname. The return statement returns a 301 permanent redirect to the HTTPS server at the same host and request URI.

解决方案

通过使用 `rewrite` 重写将所有 HTTP 请求重定向至 HTTPS:

```
server {  
    listen 80 default_server;  
    listen [::]:80 default_server;  
    server_name _;  
    return 301 https://$host$request_uri;  
}
```

`server` 块级指令配置了用于监听所有 IPv4 和 IPv6 地址的 80 端口，`return` 指令将请求及请求 URI 重定向至相同域名的 HTTPS 服务器并响应 301 状态码给客户端。

Discussion

It's important to always redirect to HTTPS where appropriate. You may find that you do not need to redirect all requests but only those with sensitive information being passed between client and server. In that case, you may want to put the return statement in particular

locations only, such as /login.

结论

在必要的场景下将 HTTP 请求重定向至 HTTPS 请求对系统安全来说很重要。有时，我们并不需要将所有的用户请求都重定向至 HTTPS 服务器，而仅需将包含用户敏感数据的请求重定向至 HTTPS 服务即可，比如用户登录服务。

20.3 HTTP Strict Transport Security

Problem

You need to instruct browsers to never send requests over HTTP.

问题

需要告知浏览器不要使用 HTTP 发送请求

Solution

Use the HTTP Strict Transport Security (HSTS) enhancement by setting the Strict-Transport-Security header:

```
add_header Strict-Transport-Security max-age=31536000;
```

This configuration sets the Strict-Transport-Security header to a max age of a year. This will instruct the browser to always do an internal redirect when HTTP requests are attempted to this domain, so that all requests will be made over HTTPS.

解决方案

通过设置 Strict-Transport-Security 响应头信息，启用 HTTP Strict Transport Security 策略，告知浏览器不支持 HTTP 请求:

```
add_header Strict-Transport-Security max-age=31536000;
```

这里，我们将 Strict-Transport-Security 消息头有效期设置为 1 年，其作用是，当用户发起一个 HTTP 请求时，浏览器在内部做一个重定向，将所有请求直接通过 HTTPS 协议访问。

Discussion

For some applications a single HTTP request trapped by a man in the middle attack could be the end of the company. If a form post containing sensitive information is sent over HTTP, the HTTPS redirect from NGINX won't save you; the damage is done. This optin security enhancement informs the browser to never make an HTTP request, therefore the request is never sent unencrypted.

结论

这是因为即使我们在服务器内部启用了 HTTPS 重定向功能，但浏览器端依然是 HTTP 请求，这可能会被中间人攻击，导致用户敏感数据泄露。这时候 HTTPS 重定向功能无法保证数据的安全性。当使用 Strict-Transport-Security 头时，浏览器将不会发送未被加密的 HTTP 请求，取而代之的是 HTTPS 请求，有效杜绝不安全的请求访问。

Also See

RFC-6797 HTTP Strict Transport Security

OWASP HSTS Cheat Sheet

[MDN HTTP Strict Transport Security](https://developer.mozilla.org/zh-CN/docs/Security/HTTP_Strict_Transport_Security)

Part III: Deployment and Operations

This is the third and final part of the NGINX Cookbook. This part will focus on deployment and operations of NGINX and NGINX Plus, the licensed version of the server. Throughout this part, you will learn about deploying NGINX to Amazon Web Services, Microsoft Azure, and Google Cloud Compute, as well as working with NGINX in Docker containers. This part will dig into using configuration management to provision NGINX servers with tools such as Puppet, Chef, Ansible, and SaltStack. It will also get into automating with NGINX Plus through the NGINX Plus API for on-the-fly reconfiguration and using Consul for service discovery and configuration templating. We'll use an NGINX module to conduct A/B testing and acceptance during deployments. Other topics covered are using NGINX's GeoIP module to discover the geographical origin of our clients, including it in our logs, and using it in our logic. You'll learn how to format access logs and set log levels of error logging for debugging. Through a deep look at performance, this part will provide you with practical tips for optimizing your NGINX configuration to serve more requests faster. It will help you install, monitor, and maintain the NGINX application delivery platform.

29.0 Introduction

Logging is the basis of understanding your application. With NGINX you have great control over logging information meaningful to you and your application. NGINX allows you to divide access logs into different files and formats for different contexts and to change the log level of error logging to get a deeper understanding of what's happening. The capability of streaming logs to a centralized server comes innately to NGINX through its Syslog logging capabilities. In this chapter, we'll discuss access and error logs, streaming over the Syslog protocol, and tracing requests end to end with request identifiers generated by NGINX.

29.0 介绍

日志记录是理解应用程序的基础。NGINX 服务器提供可控的日志记录方案，让应用的日志能更好的服务开发运维。NGINX 能够依据不同上下文环境和不同的日志等级进行记录，这能使开发及运维人员更加深入的了解服务器的处理情况。NGINX 日志记录的实现原理是使用服务器的 syslog 日志功能。本章，我们将学习访问日志、错误日志、syslog 日志协议和依据 NGINX 生成的日志标识跟踪用户请求。

30.0 Introduction

Tuning NGINX will make an artist of you. Performance tuning of any type of server or application is always dependent on a number of variable items, such as, but not limited to, the environment, use case, requirements, and physical components involved. It's common to practice bottleneck-driven tuning, meaning to test until you've hit a bottleneck, determine the bottleneck, tune for limitation, and repeat until you've reached your desired performance requirements. In this chapter we'll suggest taking measurements when performance tuning by testing with automated tools and measuring results. This chapter will also cover connection tuning for keeping connections open to clients as well as upstream servers, and serving more connections by tuning the operating system.

30.0 介绍

对 NGINX 服务器进行调优会让你成为使用 NGINX 服务器的艺术家。对服务器或应用进行性能调优影响因素颇多，包括但不限于：具体环境、用例、项目依赖和物理设备等。对项目在测试期间进行性能瓶颈测试调优十分常见，测试的目的是将服务测试直至遇到性能瓶颈、确定性能瓶颈、调优、在重复测试直至达到预期性能为止。本章，我们将学习自动化测试工具及测试结果进行优化处理；还将学习对连接的调优，以保持客户端与服务器的连接，和通过调优使服务器提供更强连接能力。

29.1 Configuring Access Logs

Problem

You need to configure access log formats to add embedded variables to your request logs.

问题

需要配置自定义格式的访问日志(access log)

Solution

Configure an access log format:

```
http {
    log_format geoproxy
        '[$time_local] $remote_addr '
        '$realip_remote_addr $remote_user '
        '$request_method $server_protocol '
        '$scheme $server_name $uri $status '
        '$request_time $body_bytes_sent '
        '$geoip_city_country_code3 $geoip_region '
        '"$geoip_city" $http_x_forwarded_for '
        '$upstream_status $upstream_response_time '
        '"$http_referer" "$http_user_agent"';
    ...
}
```

This log format configuration is named `geoproxy` and uses a number of embedded variables to demonstrate the power of NGINX logging. This configuration shows the local time on the server when the request was made, the IP address that opened the connection, and the IP of the client as NGINX understands it per `geoip_proxy` or `realip_header` instructions. `$remote_user` shows the username of the user authenticated by basic authentication, followed by the

request method and protocol, as well as the scheme, such as HTTP or HTTPS. The server name match is logged as well as the request URI and the return status code. Statistics logged include the processing time in milliseconds and the size of the body sent to the client. Information about the country, region, and city are logged. The HTTP header X-Forwarded-For is included to show if the request is being forwarded by another proxy. The upstream module enables some embedded variables that we've used that show the status returned from the upstream server and how long the upstream request takes to return. Lastly we've logged some information about where the client was referred from and what browser the client is using. The `log_format` directive is only valid within the HTTP context.

解决方案

配置访问日志格式：

```
http {
    log_format geoproxy
        '[$time_local] $remote_addr '
        '$realip_remote_addr $remote_user '
        '$request_method $server_protocol '
        '$scheme $server_name $uri $status '
        '$request_time $body_bytes_sent '
        '$geoip_city_country_code3 $geoip_region '
        '"$geoip_city" $http_x_forwarded_for '
        '$upstream_status $upstream_response_time '
        '"$http_referer" "$http_user_agent"';
    ...
}
```

这个日志配置被命名为 `geoproxy`，它使用许多 NGINX 变量来演示 NGINX 日志记录功能。接下来详细讲解配置选项中各个变量的具体含义：当用户发起请求时，会记录服务器时间(`$time_local`)、用于 NGINX 处理 `geoip_proxy` 和 `realip_header`

指令的打开连接的 IP 地址和客户端 IP 地址；`$remote_user` 值为通过基本授权的用户名；之后记录 HTTP 请求方法(`$request_method`)、协议(`$server_protocol`)和 HTTP 方法(`$scheme` : http 或 https)；当然还有服务器名称(`$server_name`)、请求的 URI 和响应状态码。除基本信息外，还有一些统计的结果数据：包括请求处理的毫秒级时间(`$request_time`)、服务器响应的数据块大小(`$body_bytes_sent`)。此外，客户端所在国家(`$geoip_city_country_code3`)、地区(`$geoip_region`)和城市信息(`$geoip_city`)也被记录在内。变量 `$http_x_forwarded_for` 用于记录由其它代理服务器发起的请求的 X-Forwarded-For 头消息。upstream 模块中一些数据也被记录到日志里：被代理服务器的响应状态码(`$upstream_status`)和服务器处理时间(`$upstream_response_time`)。请求来源(`$http_referer`)和用 户代理(`$http_user_agent`) 也有被记录在日志里。从上面可以看出 NGINX 日志记录功能还是非常强大和灵活的，不过用于定义日志格式的 `log_format` 指令仅适用于 http 块级指令内，这一点需要注意。

This log configuration renders a log entry that looks like the following:

```
[25/Nov/2016:16:20:42 +0000] 10.0.1.16 192.168.0.122 Derek
GET HTTP/1.1 http www.example.com / 200 0.001 370 USA MI
"Ann Arbor" - 200 0.001 "-" "curl/7.47.0"
```

To use this log format, use the `access_log` directive, providing a logfile path and the format name `geoproxy` as parameters:

```
server {
    access_log /var/log/nginx/access.log geoproxy;
    ...
}
```

The `access_log` directive takes a logfile path and the format name as parameters. This directive is valid in many contexts and in each context can have a different log path and or log format.

这个日志的每个日志记录结果类似下面示例：

```
[25/Nov/2016:16:20:42 +0000] 10.0.1.16 192.168.0.122 Derek  
GET HTTP/1.1 http www.example.com / 200 0.001 370 USA MI  
"Ann Arbor" - 200 0.001 "-" "curl/7.47.0"
```

如果需要使用这个日志配置，需要结合使用 `access_log` 指令，`access_log`

指令接收一个日志目录和使用的配置名作为参数：

```
server {  
    access_log /var/log/nginx/access.log geoproxy;  
    ...  
}
```

`access_log` 能在多个上下文使用，每个上下文中可以定义各自的日志目录和日志记录格式。

Discussion

The log module in NGINX allows you to configure log formats for many different scenarios to log to numerous logfiles as you see fit. You may find it useful to configure a different log format for each context, where you use different modules and employ those modules' embedded variables, or a single, catchall format that provides all necessary information you could ever want. It's also possible to structure format to log in JSON or XML. These logs will aid you in understanding your traffic patterns, client usage, who your clients are, and where they're coming from. Access logs can also aid you in finding lag in responses and issues with upstream servers or particular URIs. Access logs can be used to parse and play back traffic patterns in test environments to mimic real user interaction. There's limitless possibility to logs when troubleshooting, debugging, or analyzing your application or market.

结论

NGINX 中的日志模块允许您为不同的场景配置日志格式，以便查看不同的日志文件。

在实际运用中，为不同上下文配置不同的日志会非常有用，记录的日志内容可以简单的信息，也可以事无巨细的记录所有必要信息。不仅如此，日志内容除了支持文本也能记录 JSON 格式和 XML 格式数据。实际上 NGINX 日志有助于您了解服务器流量、客户端使用情况和客户端来源等信息。此外，访问日志还可以帮助您定位与上游服务器或特定 uri 相关的响应和问题；对于测试来讲，访问日志同样有用，它可以用于分析流量情况，模拟真实的用户交互场景。日志在故障排除、调试、应用分析及业务调整中作用是不可或缺的。

29.2 Configuring Error Logs

Problem

You need to configure error logging to better understand issues with your NGINX server.

问题

需要更深入的定位 NGINX 服务器问题，以配置错误日志。

Solution

Use the `error_log` directive to define the log path and the log level:

```
error_log /var/log/nginx/error.log warn;
```

The `error_log` directive requires a path; however, the log level is optional. This directive is valid in every context except for if statements. The log levels available are debug, info, notice, warn, error, crit, alert, or emerg. The order in which these log levels were introduced is also the order of severity from least to most. The debug log level is only available if NGINX is configured with the `--with-debug` flag.

解决方案

使用 `error_log` 指令定义错误日志目录及记录错误日志的等级:

```
error_log /var/log/nginx/error.log warn;
```

`error_log` 指令配置时需要一个必选的日志目录和一个可选的错误等级选项。

除 `if` 指令外，`error_log` 指令能在所有的上下文中使用。错误日志等级包括：

debug、info、notice、warn、error、crit、alert 和 emerg。给出的日志

等级顺序就是记录最小到最严谨的日志等级顺序。需要注意的是 debug 日志

需要在编译 NGINX 服务器时，带上 --with-debug 标识才能使用。

Discussion

The error log is the first place to look when configuration files are not working correctly. The log is also a great place to find errors produced by application servers like FastCGI. You can use the error log to debug connections down to the worker, memory allocation, client IP, and server. The error log cannot be formatted. However, it follows a specific format of date, followed by the level, then the message.

结论

请牢记当服务器配置出错时，首先需要查看错误日志以定位问题。当然，错误日志也是定温应用服务器(如 FastCGI 服务)的利器。通过错误日志，我们可以调试 worker 进程连接错误、内存分配、客户端 IP 和应用服务器等问题。错误日志格式虽然不支持自定义日志格式；但是，它同样记录当前时间、日志等级和具体信息等数据。

29.3 Forwarding to Syslog

Problem

You need to forward your logs to a Syslog listener to aggregate logs to a centralized service.

问题

需要将错误日志通过 **syslog** 服务记录到集中日志服务器。

Solution

Use the `access_log` and `error_log` directives to send your logs to a Syslog listener:

```
error_log syslog:server=10.0.1.42 debug;  
access_log syslog:server=10.0.1.42,tag=nginx,severity=info geoproxy;
```

The `syslog` parameter for the `error_log` and `access_log` directives is followed by a colon and a number of options. These options include the required server flag that denotes the IP, DNS name, or Unix socket to connect to, as well as optional flags such as `facility`, `severity`, `tag`, and `nohostname`. The `server` option takes a port number, along with IP addresses or DNS names. However, it defaults to UDP 514. The `facility` option refers to the facility of the log message defined as one of the 23 defined in the RFC standard for Syslog; the default value is `local7`. The `tag` option tags the message with a value. This value defaults to `nginx`. `severity` defaults to `info` and denotes the severity of the message being sent. The `nohostname` flag disables adding the `hostname` field

into the Syslog message header and does not take a value.

解决方案

在使用 `error_log` 和 `access_log` 指令时，将日志发送至 `syslog` 监听器：

```
error_log syslog:server=10.0.1.42 debug;  
access_log syslog:server=10.0.1.42,tag=nginx,severity=info geoproxy;
```

`error_log` 和 `access_log` 指令的 `syslog` 参数紧跟冒号(:)和一些参数选项。

包括：必选的 `server` 标记表示需要连接的 IP、DNS 名称或 UNIX 套接字；

可选参数有 `facility`、`severity`、`tag` 和 `nohostname`。 `server` 参数接收带

端口的 IP 地址或 DNS 名称；默认是 UDP 514 端口。 `facility` 参数设置

`syslog` 的类型(facility)，值是 `syslog RFC` 标准定义的 23 个值中的一个

(@todo)。 `tag` 参数表示日志文件中显示时候的标题，默认值是 `nginx`。

`severity` 设置消息严重程度，默认是 `info` 级别日志。 `nohostname` 选项，禁

止将 `hostname` 域添加到 `syslog` 的消息头中。

Discussion

Syslog is a standard protocol for sending log messages and collect-

ing those logs on a single server or collection of servers. Sending

logs to a centralized location helps in debugging when you've got

multiple instances of the same service running on multiple hosts.

This is called aggregating logs. Aggregating logs allows you to view

logs together in one place without having to jump from server to

server and mentally stitch together logfiles by timestamp. A com-

mon log aggregation stack is ElasticSearch, Logstash, and Kibana,

also known as the ELK Stack. NGINX makes streaming these logs to

your Syslog listener easy with the `access_log` and `error_log` direc-

tives.

结论

syslog 是用于在单台服务器或服务器集群中记录和收集日志的标准协议。

在多个主机上运行相同服务的多个实例时，将日志发送到集中位置有助于调试，这称为聚合日志。聚合日志允许您在一个地方查看日志，而不必切换不同服务器，并通过时间戳将日志文件集成在一起。常见聚合日志解决方案有 Elasticsearch、Logstash、Kibana 和 ELK Stack。但 NGINX 通过发送日志到 syslog 监听器，能够很容易的将 access_log 和 error_log 指令捕捉的日志发送到聚合日志服务器上。

参考

[RFC3164 - BSD Syslog 协议](<https://www.jianshu.com/p/8656fc85e497>)

[Nginx 文档-记录日志到 syslog](<https://oopsguy.com/2017/07/23/nginx-document-logging-to-syslog/>)

[关于 syslog](<http://blog.csdn.net/smstong/article/details/8919803>)

[syslog](<https://en.wikipedia.org/wiki/Syslog>)

29.4 Request Tracing

Problem

You need to correlate NGINX logs with application logs to have an end-to-end understanding of a request.

问题

需要结合 NGINX 日志和应用日志，查看请求调用栈。

Solution

Use the request identifying variable and pass it to your application to log as well:

```
log_format trace '$remote_addr - $remote_user [$time_local] '
                  '$request' $status $body_bytes_sent '
                  '$http_referer' '$http_user_agent' '
                  '$http_x_forwarded_for' $request_id';

upstream backend {
    server 10.0.0.42;
}

server {
    listen 80;

    add_header X-Request-ID $request_id; \# Return to client

    location / {
        proxy_pass http://backend;
        proxy_set_header X-Request-ID $request_id; \#Pass to app
        access_log /var/log/nginx/access_trace.log trace;
    }
}
```

In this example configuration, a `log_format` named `trace` is set up, and the variable `$request_id` is used in the log. This `$request_id` variable is also passed to the upstream application by use of the `proxy_set_header` directive to add the request ID to a header when

making the upstream request. The request ID is also passed back to the client through use of the `add_header` directive setting the request ID in a response header.

解决方案

使用 `request` 标识，并将标识写入到应用日志里：

```
log_format trace '$remote_addr - $remote_user [$time_local] '
                '$request' $status $body_bytes_sent '
                '$http_referer' '$http_user_agent' '
                '$http_x_forwarded_for' $request_id';

upstream backend {
    server 10.0.0.42;
}

server {
    listen 80;

    add_header X-Request-ID $request_id; \# Return to client

    location / {
        proxy_pass http://backend;
        proxy_set_header X-Request-ID $request_id; \#Pass to app
        access_log /var/log/nginx/access_trace.log trace;
    }
}
```

示例中，配置了名为 `trace` 的访问日志格式，并在日志中使用 `$request_id` 参数。同时，通过 `proxy_set_header` 指令将 `request` 标记(request ID)设置到请求头里，当请求匹配到 `location /` 前缀，请求被转发到 `upstream` 模块，这样同一个 `request` 标识就能记录到应用服务器日志里；此外，通过 `add_header` 指令将 `request` 标识设置到响应消息头，供客户端使用。

Discussion

Made available in NGINX Plus R10 and NGINX version 1.11.0, the `$request_id` provides a randomly generated string of 32 hexadecimal characters that can be used to uniquely identify requests. By

passing this identifier to the client as well as to the application, you can correlate your logs with the requests you make. From the front-end client, you will receive this unique string as a response header and can use it to search your logs for the entries that correspond. You will need to instruct your application to capture and log this header in its application logs to create a true end-to-end relationship between the logs. With this advancement, NGINX makes it possible to trace requests through your application stack.

结论

该功能在 NGINX Plus R10 版本和 NGINX 开源版的 1.11.0 版本可用，`$request_id` 提供了一个随机生成的 32 个十六进制字符的字符串，这些字符可以用来唯一地标识请求。通过将此标识符传递给客户端和应用服务器，可以将日志与请求关联起来。客户端会收到唯一的 `request` 标识，服务端也能使用该标识进行日志筛选。应用服务器使用时，需要获取这个消息头，以建立日志之间的关联。基于这个特性，NGINX 能够构建从客户端请求到应用服务器做出响应的整个请求调用周期的所有日志信息之间的关联。

30.0 Introduction

Tuning NGINX will make an artist of you. Performance tuning of any type of server or application is always dependent on a number of variable items, such as, but not limited to, the environment, use case, requirements, and physical components involved. It's common to practice bottleneck-driven tuning, meaning to test until you've hit a bottleneck, determine the bottleneck, tune for limitation, and repeat until you've reached your desired performance requirements. In this chapter we'll suggest taking measurements when performance tuning by testing with automated tools and measuring results. This chapter will also cover connection tuning for keeping connections open to clients as well as upstream servers, and serving more connections by tuning the operating system.

30.0 介绍

对 NGINX 服务器进行调优会让你成为使用 NGINX 服务器的艺术家。对服务器或应用进行性能调优影响因素颇多，包括但不限于：具体环境、用例、项目依赖和物理设备等。对项目在测试期间进行性能瓶颈测试调优十分常见，测试的目的是将服务测试直至遇到性能瓶颈、确定性能瓶颈、调优、在重复测试直至达到预期性能为止。本章，我们将学习自动化测试工具及测试结果进行优化处理；还将学习对连接的调优，以保持客户端与服务器的连接，和通过调优使服务器提供更强连接能力。

30.0 Introduction

Tuning NGINX will make an artist of you. Performance tuning of any type of server or application is always dependent on a number of variable items, such as, but not limited to, the environment, use case, requirements, and physical components involved. It's common to practice bottleneck-driven tuning, meaning to test until you've hit a bottleneck, determine the bottleneck, tune for limitation, and repeat until you've reached your desired performance requirements. In this chapter we'll suggest taking measurements when performance tuning by testing with automated tools and measuring results. This chapter will also cover connection tuning for keeping connections open to clients as well as upstream servers, and serving more connections by tuning the operating system.

30.0 介绍

对 NGINX 服务器进行调优会让你成为使用 NGINX 服务器的艺术家。对服务器或应用进行性能调优影响因素颇多，包括但不限于：具体环境、用例、项目依赖和物理设备等。对项目在测试期间进行性能瓶颈测试调优十分常见，测试的目的是将服务测试直至遇到性能瓶颈、确定性能瓶颈、调优、在重复测试直至达到预期性能为止。本章，我们将学习自动化测试工具及测试结果进行优化处理；还将学习对连接的调优，以保持客户端与服务器的连接，和通过调优使服务器提供更强连接能力。

30.1 Automating Tests with Load Drivers | 使用负载工具实现自动化测试

Problem

You need to automate your tests with a load driver to gain consistency and repeatability in your testing.

问题

使用负载测试工具实现自动化测试

Solution

Use an HTTP load testing tool such as Apache JMeter, Locust, Gatling, or whatever your team has standardized on. Create a configuration for your load-testing tool that runs a comprehensive test on your web application. Run your test against your service. Review the metrics collected from the run to establish a baseline. Slowly ramp up the emulated user concurrency to mimic typical production usage and identify points of improvement. Tune NGINX and repeat this process until you achieve your desired results.

解决方案

使用 HTTP 负载测试工具：如 Apache JMeter/ Locust/ Gatling/或团队自研的负载工具。为测试定制测试配置，并对服务器进行全面测试，基于测试结果量化性能指标；之后，逐步增加用户数增加并发量，以模拟生产环境真实请求，找出性能瓶颈优化；如此反复，直至达到项目的预期性能。

Discussion

Using an automated testing tool to define your test gives you a consistent test to build metrics off of when tuning NGINX. You must be able to repeat your test and measure performance gains or losses to conduct science. Running a test before making any tweaks to the NGINX configuration to establish a baseline gives you a basis to work from so that you can measure if your configuration change has improved performance or not. Measuring for each change made will help you identify where your performance enhancements come from.

结论

使用自动化的测试工具来定义您的测试，可以让您通过一个一致的测试来找出对 NGINX 进行调优的基准。性能测试必须是可重复的，通过对性能测试结果进行科学分析。在对 NGINX 配置进行优化前，需对服务器进行测试确定标准，这样，才能确定之后的配置优化是否实现了性能的优化。对每个配置优化进行度量，将帮助您确定性能得以提升的根源。

30.2 Keeping Connections Open to Clients

Problem

You need to increase the number of requests allowed to be made over a single connection from clients and the amount of time idle connections are allowed to persist.

问题

增加单个连接的请求数，同时增加空闲连接(idle connections)的连接时长。

Solution

Use the `keepalive_requests` and `keepalive_timeout` directives to alter the number of requests that can be made over a single connection and the time idle connections can stay open:

```
http {
    keepalive_requests 320;
    keepalive_timeout 300s;
    ...
}
```

The `keepalive_requests` directive defaults to 100, and the `keepalive_timeout` directive defaults to 75 seconds.

解决方案

`keepalive_requests` 和 `keepalive_timeout` 指令允许变更单个连接的最大请求数和空闲连接的连接时长：

```
http {
    keepalive_requests 320;
    keepalive_timeout 300s;
    ...
}
```


`keepalive_requests` 默认为 100，`keepalive_timeout` 的默认值为 75 秒。

Discussion

Typically the default number of requests over a single connection will fulfill client needs because browsers these days are allowed to open multiple connections to a single server per fully qualified domain name. The number of parallel open connections to a domain is still limited typically to a number less than 10, so in this regard, many requests over a single connection will happen. A trick commonly employed by content delivery networks is to create multiple domain names pointed to the content server and alternate which domain name is used within the code to enable the browser to open more connections. You might find these connection optimizations helpful if your frontend application continually polls your backend application for updates, as an open connection that allows a larger number of requests and stays open longer will limit the number of connections that need to be made.

结论

一般情况下，`keepalive_requests` 和 `keepalive_timeout` 的默认配置，能够满足客户端的请求，因为，现代浏览器能为不同域名打开多个连接。但对于同一个域名仅能同时发起 10 以内的请求，这将带来性能瓶颈。CDN 的实现原理是启用多个域名指向内容服务器，并以编码的方式指定使用的域名，以使浏览器能够打开更多的连接。你会发现使用更多的请求连接数和连接时长配置，在客户端需要频繁更新数据能提升服务器性能。

30.3 Keeping Connections Open Upstream

Problem

You need to keep connections open to upstream servers for reuse to enhance your performance.

问题

需要增加代理服务器与被代理服务器的连接数，提升服务器性能。

Solution

Use the `keepalive` directive in the upstream context to keep connections open to upstream servers for reuse:

```
proxy_http_version 1.1;
proxy_set_header Connection "";
upstream backend {
    server 10.0.0.42;
    server 10.0.2.56;
    keepalive 32;
}
```

The `keepalive` directive in the upstream context activates a cache of connections that stay open for each NGINX worker. The directive denotes the maximum number of idle connections to keep open per worker. The proxy modules directives used above the upstream block are necessary for the `keepalive` directive to function properly for upstream server connections. The `proxy_http_version` directive instructs the proxy module to use HTTP version 1.1, which allows for multiple requests to be made over a single connection while it's open. The `proxy_set_header` directive instructs the proxy module to strip the default header of `close`, allowing the connection

to stay open.

解决方案

在 upstream 会计指令中使用 **keepalive** 指令保持代理服务与被代理服务器连接以复用：

```
proxy_http_version 1.1;
proxy_set_header Connection "";
upstream backend {
    server 10.0.0.42;
    server 10.0.2.56;
    keepalive 32;
}
```

keepalive 指令会为每个 NGINX worker 进程创建一个连接缓存，表示每个 worker 进程能保持打开的空闲连接的最大连接数量。如果要使 **keepalive** 指令正常工作，在 upstream 指令上使用的 proxy 模块指令则是必须的。
proxy_http_version 指令表示启用的 http 1.1 版本，它允许在单个连接上发送多个请求；**proxy_set_header** 指令删除 connection 消息头的默认值 **close**，这样就允许保持连接的打开状态。

Discussion

You would want to keep connections open to upstream servers to save the amount of time it takes to initiate the connection, and the worker process can instead move directly to making a request over an idle connection. It's important to note that the number of open connections can exceed the number of connections specified in the **keepalive** directive as open connections and idle connections are not the same. The number of **keepalive** connections should be kept small enough to allow for other incoming connections to your upstream server. This small NGINX tuning trick can save some

cycles and enhance your performance.

结论

当需要保持代理服务器与被代理服务器的连接打开状态，以节省启动连接所需的时间；同时，需要将 **worker** 进程收到的请求分发值空闲的连接直接处理。有一点需要注意，开启的连接数可以多于 **keepalive** 配置的连接数，因为开启的连接数和空闲连接数不是同一个东西。**keepalive** 配置的连接数应尽量少，以确保新的请求能够被分发到被代理服务器。这条配置技巧能够通过减少请求连接的生命周期的手段，提升服务器性能。

30.4 Buffering Responses

Problem

You need to buffer responses between upstream servers and clients in memory to avoid writing responses to temporary files.

问题

将服务器对客户端的响应写入内存缓冲区而不是文件里。

Solution

Tune proxy buffer settings to allow NGINX the memory to buffer response bodies:

```
server {
    proxy_buffering on;
    proxy_buffer_size 8k;
    proxy_buffers 8 32k;
    proxy_busy_buffer_size 64k;
    ...
}
```

The `proxy_buffering` directive is either on or off; by default it's on.

The `proxy_buffer_size` denotes the size of a buffer used for reading the first part of the response from the proxied server and defaults to either 4k or 8k, depending on the platform. The `proxy_buffers` directive takes two parameters: the number of buffers and the size of the buffers. By default the `proxy_buffers` directive is set to a number of 8 buffers of size either 4k or 8k, depending on the platform. The `proxy_busy_buffer_size` directive limits the size of buffers that can be busy, sending a response to the client while the response is not fully read. The busy buffer size defaults to

double the size of a proxy buffer or the buffer size.

解决方案

调整代理模块的缓存区设置，允许 NGINX 服务器将响应消息体写入内存缓冲区：

```
server {  
    proxy_buffering on;  
    proxy_buffer_size 8k;  
    proxy_buffers 8 32k;  
    proxy_busy_buffer_size 64k;  
    ...  
}
```

`proxy_buffering` 值可以使 `on` 或 `off`，默认是 `on`。`proxy_buffer_size` 指令

表示用于读取来自代理服务器响应的缓冲大小，依据平台不同它的默认值为

4k 或 8k。`proxy_buffers` 指令包含两个值，支持的缓存区个数和单个缓存区

容量大小，默认是 8 个缓存区，依据平台不同单个缓存区默认容量为 4k 或 8k。

`proxy_busy_buffer_size` 指令用于配置未完全读取响应时直接响应客户端的缓冲

区大小，它的空间一般为 `proxy_buffers` 的两倍为最佳。

Discussion

Proxy buffers can greatly enhance your proxy performance, depending on the typical size of your response bodies. Tuning these settings can have adverse effects and should be done by observing the average body size returned, and thoroughly and repeatedly testing.

Extremely large buffers set when they're not necessary can eat up the memory of your NGINX box. You can set these settings for specific locations that are known to return large response bodies for optimal performance.

结论

代理缓存能显著提升代理服务性能，这取决于响应内容的大小。开启缓冲区设置应当仔细测试响应内容的平均大小，并进行大量测试和调试，否则可能引发副作用。而如果将缓存区大小设置的非常大也不行，这回占用大量的 NGINX 内存。一种方案是将缓冲区大小设置为与最大响应消息相同以提升性能。

参考

[nginx缓冲区优化](<http://www.cnblogs.com/me115/p/5698787.html>)

30.5 Buffering Access Logs

Problem

You need to buffer logs to reduce the opportunity of blocks to the NGINX worker process when the system is under load.

问题

当系统处于负载状态时，启用日志缓冲区以降低 NGINX worker 进程阻塞。

Solution

Set the buffer size and flush time of your access logs:

```
http {  
    access_log /var/log/nginx/access.log main buffer=32k flush=1m;  
}
```

The buffer parameter of the access_log directive denotes the size of a memory buffer that can be filled with log data before being written to disk. The flush parameter of the access_log directive sets the longest amount of time a log can remain in a buffer before being written to disk.

解决方案

设置 access_log 的 buffer 和 flush 参数：

```
http {  
    access_log /var/log/nginx/access.log main buffer=32k flush=1m;  
}
```

buffer 参数用于设置，写入文件前的缓冲区内存大小；flush 参数设置缓冲区内日志在缓冲区内存中保存的最长时间。

Discussion

Buffering log data into memory may be a small step toward optimization. However, for heavily requested sites and applications, this can make a meaningful adjustment to the usage of the disk and CPU. When using the `buffer` parameter to the `access_log` directive, logs will be written out to disk if the next log entry does not fit into the buffer. If using the `flush` parameter in conjunction with the `buffer` parameter, logs will be written to disk when the data in the buffer is older than the time specified. When buffering logs in this way, when tailing the log, you may see delays up to the amount of time specified by the `flush` parameter.

结论

将日志数据缓冲到内存中可能是很小的一个优化手段。但是，对于有大量请求的站点和应用程序的磁盘读写和 CPU 使用性能有重大意义。`buffer` 参数的功能是当缓冲区已经写满时，日志会被写入文件中；`flush` 参数的功能是，当缓存中的日志超过最大缓存时间，也会被写入到文件中，不过也有不足的地方即写入到日志文件的日志有些许延迟。

